INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Frameworks e Bibliotecas Javascript

NUNO FILIPE BRANDAO DUARTE
Outubro de 2015

POLITÉCNICO
DO PORTO

Frameworks e Bibliotecas JavaScript

Nuno Filipe Brandão Duarte

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquitecturas Sistemas e Redes

Orientador: Nuno Silva
Co-orientador: Paulo Maio

Porto, Outubro 2015

Abstract

This report describes a study based on a research and evaluation of platforms (JavaScript frameworks and libraries carried out under the discipline of Tese/Dissertação/Estágio in the Master Degree Computer Science - area of expertise in Arquitecturas Sistemas e Redes, in Instituto Superior de Engenharia do Porto (ISEP).

The JavaScript language is increasingly present in the development of software and particularly software for the web environment.

For this reason there is a need to investigate the issue, as the platforms that are emerging in the market have a very low level of maturity compared to existing, it is crucial to identify the main differences so that we can save time, labor and other costs in research and experimentation developers.

This project aims to inform a range of platforms and existing JavaScript libraries on the market.

For further investigation, a study of use cases used at this point in the market was held.

To assist the work were used qualitative research methods through the existing content of research related to the area, and quantitative research methods.

It was thus conducted a survey sent via email to fifty regular users of the language, platforms, and JavaScript libraries, linking the functional and non-functional characteristics of the platforms and existing use cases in the market, in order to realize what is the use of these tools in his professional life.

This study allowed developers to access information that will be compared and evaluated in order to achieve a more precise evaluation of the various analyzed platforms.

It was also made an evaluation of adaptation of the various platforms to the various use cases.

Key Words: Platforms, Frameworks, Libraries, Javascript, Use cases

Resumo

O presente relatório descreve um estudo de pesquisa e avaliação de plataformas (frameworks e bibliotecas JavaScript realizado no âmbito da disciplina de Tese/Dissertação/Estágio do Mestrado em Engenharia Informática - área de especialização de Arquitecturas Sistemas e Redes, do Instituto Superior de Engenharia do Porto (ISEP).

A linguagem JavaScript está cada vez mais presente no desenvolvimento de software e em particular de software para ambiente web.

Por esse motivo surge a necessidade de se investigar sobre o tema, visto que as plataformas que vão surgindo no mercado têm um nível de maturidade muito reduzido comparativamente às já existentes, sendo crucial identificar as suas principais diferenças para que se possa poupar tempo, trabalho e outros custos na pesquisa e experimentação aos desenvolvedores.

Este projeto tem como objetivo dar a conhecer um conjunto de plataformas e bibliotecas JavaScript existentes no mercado.

Para uma investigação mais aprofundada, foi realizado um estudo relativo aos cenários usados neste momento no mercado. Com vista a auxiliar o trabalho foram utilizados métodos de pesquisa para encontrar informação relevante para o estudo, através da pesquisa de conteúdos já existentes relacionados com a área, e métodos quantitativos de pesquisa.

Foi assim realizado um inquérito enviado via correio eletrónico para cerca de cinquenta habituais utilizadores da linguagem, plataformas e bibliotecas JavaScript, relacionando as características funcionais e não funcionais das plataformas e os cenários existentes no mercado, com o intuito de perceber qual a utilidade destas ferramentas na sua vida profissional.

Este estudo permitirá aos desenvolvedores o acesso a informação que será comparada e avaliada, com vista a atingir uma avaliação mais precisa das várias plataformas analisadas.

Faz-se além disso uma avaliação das várias plataformas aos vários cenários.

Palavras Chave: Plataformas, Bibliotecas, JavaScript, Cenários

Agradecimentos

Agradeço a todos aqueles que de forma direta ou indireta me ajudaram na concretização desta tese.

Devo também agradecer a todos os meus familiares pelo apoio incondicional e a minha namorada por toda a paciência e disponibilidade para me ajudar a ultrapassar as dificuldades.

Por fim, mas não por último, agradeço aos meus orientadores Nuno Silva e Paulo Maio, por todo apoio e por toda a ajuda que me deram, para além de toda a disponibilidade que demonstraram sempre que necessitei de algum tipo de esclarecimento.

Índice

Índice de Figuras

Índice de tabelas

Acrónimos

REST	*Representational State Transfer*
XML	*Extensible Markup Language*
JSON	*JavaScript Object Notation*
DOM	*Document Object Model*
SEO	*Search Engine Optimization*
DDP	*Distributed Data Protocol*
AJAX	*Asynchronous Javascript and XML*
SPA	*Single Page Application*
HTML5	*HyperText Markup Language 5*

1 Introdução

No âmbito da disciplina de Tese/Dissertação/Estágio do Mestrado em Engenharia Informática - área de especialização de Arquitecturas Sistemas e Redes do Instituto Superior de Engenharia do Porto, propus-me a realizar uma tese relacionada com plataformas.
Plataformas são um conjunto de frameworks e bibliotecas JavaScript. Neste contexto, entende-se por plataforma JavaScript uma framework ou biblioteca escrita em JavaScript e que pode ser utilizada no desenvolvimento de *software* também em JavaScript (na secção 2.7 Framework vs Biblioteca será debatida a diferença entre Framework e biblioteca).
Para melhor compreensão deste projeto é necessário que o leitor tenha em conta as seguintes convenções.

1. O leitor tem que ter conhecimentos básicos sobre JavaScript e todos os conceitos associados, por exemplo, HTML5, AJAX, DOM e ECMAScript.
2. Conhecimentos básicos sobre navegadores de internet e as diferenças entre eles.

Durante este estudo será feita uma classificação de vários cenários (capítulo 3 Cenários de Desenvolvimento *Web*) e plataformas (capítulo 4 Classificação) existentes no mercado.
As plataformas abordadas ao longo do documento também serão comparadas, no capítulo 5, relativamente às suas características funcionais aos cenários.
Para finalizar esta pesquisa, haverá um último capítulo (capítulo 7 Conclusões) que definirá quais as plataformas que se adaptam aos vários cenários estudados neste documento, através da realização e interpretação de um inquérito (Capítulo 6 Avaliação).

1.1 Contexto

Com a evolução do HTML5 e com o aumento do uso da *Cloud* , têm vindo a aparecer inúmeras frameworks e bibliotecas JavaScript [1]. O navegador de internet está em constante desenvolvimento proporcionando aos seus utilizadores ferramentas online, que antes se encontravam apenas num computador (e.g. Word online).
Estas plataformas melhoram progressivamente a experiência do utilizador e tornam os serviços mais rápidos e com maior facilidade de acesso.
Em suma, com o apoio das grandes empresas, como por exemplo a Google, a Microsoft e o Facebook, estas tecnologias encontram-se em constante desenvolvimento, tornando-as mais apelativas e mais simples de utilizar [2].

1.2 Problema

Devido ao aumento crescente de plataformas que têm vindo a aparecer nos últimos tempos, os desenvolvedores deparam-se com um constante problema para escolher qual a melhor plataforma para o seu trabalho [2].

Todas as plataformas têm o um conjunto específico de características funcionais que muitas vezes são executadas de maneira diferente para atingir um resultado semelhante.
Devido a serem plataformas bastante recentes, estas tendem a ter pouca maturidade e as suas comunidades são normalmente reduzidas.
Para além de tudo isto, existe ainda um problema de sobreposição de funcionalidades entre estas plataformas, isto significa que duas plataformas diferentes podem não ser compatíveis entre si.
Em suma, existe um enorme número de plataformas disponíveis que têm diferentes características e a sua maturidade e comunidade são muitas vezes reduzidas.

1.3 Motivações

Estas framewoks e bibliotecas têm como principal objetivo melhorar a experiência do utilizador e com isso aumentar as receitas das empresas.
Apesar de os *stakeholders* supracitados serem beneficiados por estas plataformas, o intuito desta pesquisa é essencialmente ajudar os desenvolvedores a escolher a melhor plataforma para os projetos que poderão vir a desenvolver.
A ideia de desenvolver um estudo nesta área surgiu quando se deu a divulgação da plataforma NodeJs e posteriormente das frameworks isomórficas.
Grande parte deste estudo baseia-se no desenvolvimento do lado do servidor de muitas aplicações *web*, conceitos como reutilização de código e performance são questões que considero relevantes para a área, sendo que com estas plataformas a reutilização de código e a performance são muito favoráveis.
Em suma, pretende-se proporcionar à comunidade que desenvolve nesta área uma ferramenta que lhes torne o processo de decisão mais simples e eficaz, evitando perdas de tempo em pesquisas relacionadas com as plataformas que mais se adequem ao seu trabalho.

1.4 Objetivos

Nesta tese irá realizar-se, através de métodos de pesquisa, uma caracterização dos diferentes cenários que existem no mercado.

Também se pretende desenvolver um sistema de categorização relativo às vantagens de cada plataforma, permitindo aos desenvolvedores reconhecer e escolher, usando exemplos práticos, a melhor plataforma para o projeto que querem desenvolver.
É também uma meta para esta pesquisa, sumarizar todas as informações relativas às várias plataformas pesquisadas, apresentando vários cenários.

1.5 Metodologia

Para mostrar as vantagens de cada uma das plataformas, tornou-se necessário perceber o que os desenvolvedores procuram nestas ferramentas.

Será assim necessário recolher e classificar as plataformas relativamente à sua sobreposição, complementaridade funcional e não funcional, permitindo assim saber quais são as bibliotecas compatíveis com cada plataforma e como o uso de uma plataforma pode impedir o funcionamento de outra.

No caso de complementaridade não funcional, será abordada a comunidade que cada plataforma tem e o seu impacto na internet.

Para isso foi realizada alguma investigação na área, através da análise de obras relacionadas com o tema, citadas ao longo da pesquisa (métodos qualitativos de investigação), e foi também realizado um inquérito (método quantitativo de investigação) com vista a compreender a importância e o funcionamento das plataformas e casos de uso mencionados ao longo do presente documento.

O documento adotará um exemplo, explicado nas convenções, que será usado ao longo dos vários capítulos com vista a expor caraterísticas dos cenários de utilização, potenciar a descrição das tecnologias analisadas e facilitar a compreensão do leitor, permitindo-lhe assim entender quais são as vantagens das plataformas aqui assinaladas.

1.6 Estrutura do documento

O presente documento vai estar estruturado da seguinte forma:

No segundo capítulo constará uma explicação relativamente às várias tecnologias e mitologias existentes neste momento na linguagem javascript. Será possível verificar como foi criado o javascript e porque é que a evolução desta linguagem de programação levou à necessidade de existirem plataformas que permitam melhorar o desempenho do desenvolvedor e aumentar a experiência do utilizador numa aplicação *web*.

No capítulo seguinte irão ser abordadas os cenários, o que permitirá perceber de uma forma mais simples o que o mercado está a procura.

No quarto capítulo encontrasse a classificação as características importantes para o desenvolvimento *web*. Este capítulo também conterá uma comparação geral das várias plataformas mostrando que características cada plataforma tem ou pode ter.

Na comparação reúne-se um conjunto de tabelas onde será possível comparar as seguintes informações: características de todas as plataformas estudadas, as características dos cenários usados neste estudo e as plataformas para os vários cenários estudados.

No quinto capítulo constará todas as informações relativamente ao inquérito realizado, sendo possível verificar as seguintes informações: formato do inquérito, análise de resultados e uma análise subjetiva do autor.
O último capítulo encontra-se divididos por diferentes cenários, para cada um desses cenários será possível verificar uma avaliação com base na informação pesquisada e também baseada no inquérito realizado. Também neste capítulo mostra-se os aspetos positivos e negativos que ocorreram ao longo desta pesquisa e identificando também possíveis melhoramentos para este trabalho.

2 Estado da arte

Neste capítulo será apresentado o estado-da-arte de bibliotecas e frameworks JavaScript.
Inicia-se com o desenvolvimento de aplicações *web*, de seguida irá abordar-se a história, evolução e caraterização do JavaScript ao longo dos anos, assim como diversos conceitos relacionados, importantes para a contextualização das bibliotecas e frameworks a analisar.
Mais à frente serão identificadas as grandes diferenças entre uma biblioteca e uma framework, bem como se elabora sobre as razões para as plataformas existirem e serem usadas.
Neste contexto torna-se necessário descrever o princípio de "inversão de controlo".

2.1 Desenvolvimento de Aplicações *Web*

O desenvolvimento de aplicações *web* consiste na construção de dois grandes componentes:

- Cliente
- Servidor

No lado do cliente são desenvolvidas páginas através da utilização de linguagens de programação como HTML, CSS e JavaScript. Estas linguagens são interpretadas no navegador de internet do utilizador da aplicação.
Por sua vez, no servidor é desenvolvida a lógica de negócio, usando por exemplo linguagens como PHP ou .NET.
Um dos conceitos importantes no desenvolvimento de aplicações web é o SEO (Search Engine Optimization) [3].
Todos os grandes motores de busca (e.g. Google e Bing) têm os seus *crawlers* (algoritmos de pesquisa), estes colecionam informação sobre todas as páginas na internet com o objetivo de ajudar os utilizadores a encontrar facilmente o que procuram. Este conceito também é importante para o desenvolvimento de aplicações *web*, porque estes sistemas têm um *raking* e permitem que mais pessoas cheguem à aplicação *web* desenvolvida.
Para melhorar a experiência para o utilizador da aplicação, foi criada a tecnologia AJAX. Esta tecnologia permite ao desenvolvedor fazer pedidos ao servidor, sem o conhecimento do

utilizador da aplicação, com vista a receber novas informações e através delas tornar a página mais dinâmica [4].
Por fim, para aumentar a interatividade da aplicação para o utilizador e para reduzir pedidos feitos ao servidor desenvolveu-se o conceito de SPA (Single page applications), conceito que será retratado com melhor detalhe no capítulo 3.1 deste documento.

2.2 JavaScript

O JavaScript foi desenvolvido inicialmente por Brendan Eich, com o nome de código "mocha", para a Netscape. Quando esta linguagem foi lançada em Setembro de 1995, o livescript foi o seu nome oficial [5].

Ainda nesse ano o departamento de marketing da Netscape queria anunciar a parceria feita com a Sun Microsystems, empresa criadora da linguagem de programação JAVA. Com vista a aproveitar toda publicidade/notícias na imprensa social, o departamento resolveu alterar o nome de livescript para JavaScript [5]. Este novo nome gerou bastante confusão pois cria a ilusão de que esta nova linguagem é baseada em JAVA, o que não é verdade [5].

Devido ao grande sucesso do JavaScript 1.0 o Netscape lançou a versão 1.1 para o navegador "Netscape Navigator 3". Tornando-se desta forma a empresa que estava à frente no mercado. Nesta altura, a Microsoft decidiu aumentar os recursos para o seu internet Explorer e implementar a sua versão do JavaScript que deu o nome de Jscript [5].

Existiam portanto duas linguagens de programação com intuitos idênticos, muito semelhantes na forma, mas com diversas incompatibilidades [6]. De facto, como ainda não eram *standard*, não haviam regras e funcionalidades pré-estabelecidas, o que causou alguns problemas levando a Netscape juntamente com outras empresas a apresentar à Organização internacional para Padronização e a Comissão eletrotécnica internacional um padrão JavaScript com o nome de ECMAScript (lê-se "ek-ma-script") [6].

2.3 Caraterísticas do JavaScript

O JavaScript é uma linguagem de programação que permite criar aplicações que proporcionam uma maior interatividade com o utilizador [5].
Algumas das caraterísticas principais do JavaScript são as seguintes:

- Linguagem interpretada: Algumas linguagens precisam de ser compiladas para código máquina, antes que possam ser executadas. No entanto o JavaScript tem uma linguagem que não precisa de ser compilada, sendo que é uma linguagem interpretada e o navegador de internet executa cada linha quando o pedido chega ao navegador de internet.

- Baseada em protótipos [7]: Esta característica é um estilo de linguagem de programação orientada a objetos, sendo também um processo de copiar objetos existentes que servem de protótipos.
- Funções em primeiro lugar: Esta característica permite ao desenvolvedor passar como argumento de funções a outras funções e guardar em variáveis essas funções.
- Tipagem fraca: As variáveis não têm que ser declaradas e não ficam restritas a um tipo (*int* ou *string*) e podem mudar durante a execução.

2.4 Evolução do JavaScript

A linguagem JavaScript sofreu várias alterações ao longo dos anos.

Como indicado no secção anterior, JavaScript era inicialmente uma linguagem que permitia criar aplicações que proporcionavam uma maior interatividade com o utilizador.

Esta linguagem confere ao desenvolvedor grandes vantagens pois permite um desenvolvimento de páginas *web* mais simples e eficaz [6].

Ao longo dos anos houve inúmeras melhorias no JavaScript como, por exemplo, a funcionalidade de fazer pedidos assíncronos ao servidor com o nome de AJAX (Asynchronous JavaScript and XML). Esta funcionalidade permite aumentar a interatividade da página através de:

- Pedidos feitos ao servidor sem carregar uma nova página ou recarregar a mesma página,
- Validação de dados,
- Atualização de informação.

Apesar de todas as suas mais-valias, detetaram-se alguns problemas no JavaScript, pois não havia um compilador universal, devido ao facto de ser uma linguagem para correr no navegador de internet do utilizador. Este problema, aliado ao facto de existirem várias versões de vários "fabricantes", tornaram esta linguagem ainda mais complicada.

Por exemplo, existiam navegadores de internet que tinham um compilador de JavaScript mais avançado [8], o que permitia ao desenvolvedor tirar maior proveito da linguagem.

No entanto, existem dois problemas relacionados com esta linguagem:

- O primeiro relaciona-se com a diferença da versão do JavaScript utilizado pela aplicação *web* e o navegador de internet
- O segundo é relativo à falta de um compilador universal, criando a necessidade de se programar tendo em conta a compatibilidade dos vários navegadores.

Para solucionar o problema de compatibilidade dos navegadores e para que o JavaScript não ficasse mais lento em alguns navegadores de internet, foram desenvolvidas várias bibliotecas que para além de virem solucionar os problemas mencionados, também permitiram libertar

tempo ao desenvolvedor, não sendo assim necessário voltar a escrever sempre o mesmo código (DRY - Don't Repeat Yourself) [9].

Uma das bibliotecas que teve e ainda tem bastante sucesso no desenvolvimento *web* é a biblioteca JQuery [1], porque trouxe maior compatibilidade entre navegadores de internet, trazendo também um leque de funcionalidades das quais o desenvolvedor pode tirar partido para o seu trabalho.

Na Figura 1 encontra-se representado código de um pedido AJAX realizado apenas em JavaScript.

```
function loadXMLDoc() {
    var xmlhttp;

    if (window.XMLHttpRequest) {
        // code for IE7+, Firefox, Chrome, Opera, Safari
        xmlhttp = new XMLHttpRequest();
    } else {
        // code for IE6, IE5
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }

    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == XMLHttpRequest.DONE ) {
            if(xmlhttp.status == 200){
                document.getElementById("myDiv").innerHTML = xmlhttp.responseText;
            }
            else if(xmlhttp.status == 400) {
                alert('There was an error 400')
            }
            else {
                alert('something else other than 200 was returned')
            }
        }
    }

    xmlhttp.open("GET", "AJAX_info.txt", true);
    xmlhttp.send();
}
```

Figura 1 - JavaScript AJAX

A Figura 2 apresenta o mesmo pedido AJAX que a Figura 1, no entanto usa a biblioteca JQuery.

```
$.ajax({
    method: "GET",
    url: "AJAX_info.txt",
    context: document.body,
    success: function(responseText){
        document.getElementById("myDiv").innerHTML = responseText;
    }
});
```

Figura 2 - JQuery AJAX

Pode-se assim comprovar que o uso de uma biblioteca JQuery permite libertar tempo ao desenvolvedor.

Consequentemente, foram desenvolvidas frameworks que são estruturas de *software* robustas que têm como principal objetivo proporcionar ao utilizador uma agradável experiência no site ao mesmo tempo que facilita o desenvolvimento de software.
Foram também identificadas várias funcionalidades que são normalmente utilizadas em grande parte dos *websites*, com vista a mais uma vez libertar tempo ao desenvolvedor, algumas dessas funcionalidades são as seguintes:

- Two way binding: Sincroniza um objeto com a representação do mesmo.
- Routing: Possibilita que o utilizador guarde as ações realizadas até ao estado atual da informação.
- Suporte mínimo dos navegadores: Verifica qual é a versão mínima do navegador de internet.

O JavaScript tem evoluído e neste momento já existem aplicações programadas principalmente em JavaScript que já não estão dentro de um navegador de internet mas sim instaladas no sistema operativo, havendo frameworks desenvolvidas principalmente para esse trabalho, por exemplo AppJs [10]. Este tipo de funcionalidades permite ao desenvolvedor portabilidade do *software* desenvolvido entre diversos sistemas [10].

2.5 Padrões de Arquitectura

Estas plataformas são normalmente desenvolvidas segundo padrões arquiteturais que permitem estruturar e desenvolver código para haver uma separação entre as várias partes que o integram.

2.5.1 Padrão de software

Um padrão é um documento escrito que descreve uma solução geral para um problema de *design* que ocorre repetidamente em vários projetos. Os *designers* de *software* adaptam a solução do padrão para o seu projeto específico [11].
Alguns dos princípios existentes nestes padrões são o *General Responsibility Assignment Software Patterns* (GRASP), o SOLID (vs. STUPID).
O GRASP é um conjunto de princípios que têm como objetivo avaliar as classes e os objetos de uma linguagem de programação orientada a objetos. Dentro desta norma existem um conjunto de diretivas que serão de seguida citadas de forma breve [12]:

- Controller: Permite designar a responsabilidade de lidar com os eventos do sistema a uma classe sem interface de utilizador.
- Creator: É uma das atividades mais comuns nos sistemas orientados a objetos. A classe que tem a responsabilidade de criar objetos é fundamental para a relação entre objetos.

- High Cohesion: Tenta manter os objetos focados numa tarefa de forma apropriada e compreensível.
- Indirection: Entre dois elementos desenvolver um intermediário que comunica com os elementos e assim permite a reutilização de código.
- Information Expert: É o princípio usado para determinar onde delegar responsabilidades (métodos, propriedades, entre outros).
- Low Coupling: É um padrão de avaliação que atribui responsabilidades de apoio:
    - Menor dependência entre as classes
    - Mudanças de uma classe devem ter o menor impacto possível sobre as outras classes.
    - Elevado potencial de reutilização
- Polymorphism: Responsabilidade de definir a variação de um comportamento baseado no tipo.
- Protected Variations: Padrão que protege as variáveis de um elemento criando uma interface entre as variáveis e os elementos que querem aceder a essas variáveis.
- Pure Fabrication: É uma classe que não representa um conceito para o domínio do problema, é realizada para atingir um baixo acoplamento, alta coesão e reutilizar o potencial derivado daí.

SOLID vs STUPID são princípios que visam guiar o desenvolvedor no sentido de programar da forma correta [9]. SOLID é um conjunto de princípios de *design*, introduzido por Robet C. Martin. Alguns desses princípios são os seguintes:

- Single Responsibility Principle: Cada Classe deve ter uma única responsabilidade. Não deve haver mais que uma razão para alterar a classe.
- Open/Closed Principle: As identidades de *software* devem ser abertas para extensão mas fechadas para modificação.
- Liskov Substitution Principle: Os objetos de uma aplicação devem ser substituíveis com instâncias dos seus subtipos sem fazer alterações na aplicação.
- Interface Segregation Principle: Ter várias interfaces específicas é melhor do que ter uma única interface para tudo.
- Dependency Inversion Principle: As abstrações não devem depender de detalhes por sua vez os detalhes não devem depender de abstrações. Por outras palavras deve-se manter o mesmo nível de abstração em todos os casos.

Por sua vez, STUPID é um conjunto de erros normalmente cometidos pelos desenvolvedores. Este último princípio visa que o desenvolvedor tenha conhecimento acerca dos erros cometidos com vista a não os produzir.

- Singleton: Padrão normalmente usado fora do contexto para qual foi criado e que aumenta o acoplamento, normalmente é considerado um anti padrão.
- Tight Coupling: Deve reduzir o acoplamento entre os módulos da aplicação.
- Untestability: Deve escrever testes unitários para verificação do código.
- Premature Optimization: Deve evitar otimizações prematuras.
- Indescriptive Naming: Deve escrever as classes, funções e atributos para humanos, sem abreviações.
- Duplication: Não se repita (DRY – Don't repeat yourself), escreva código apenas uma vez.

Para além das diretivas supracitadas existem ainda outros conjuntos de padrões como o Gang of Four (GoF) [13], os Patterns of Enterprise Application Architecture [14], entre outros.
As plataformas JavaScript adotam geralmente dois padrões arquiteturais: MVC e MVVM.

2.5.2 MVC

O padrão Model-View-Controller [13] é muito provavelmente o padrão mais utilizado em aplicações *Web*. As vantagens de utilização deste padrão são as seguintes:

- Facilitar o reaproveitamento de código;
- Facilitar adição de recursos;
- Maior integração da equipa e /ou divisão de tarefas;
- Manter o código sempre limpo;

Isto irá tornar o código mais estruturado e com isso mais simples de manter, testar e alterar.
Este padrão permite dividir a aplicação em três grandes componentes (Figura 3):

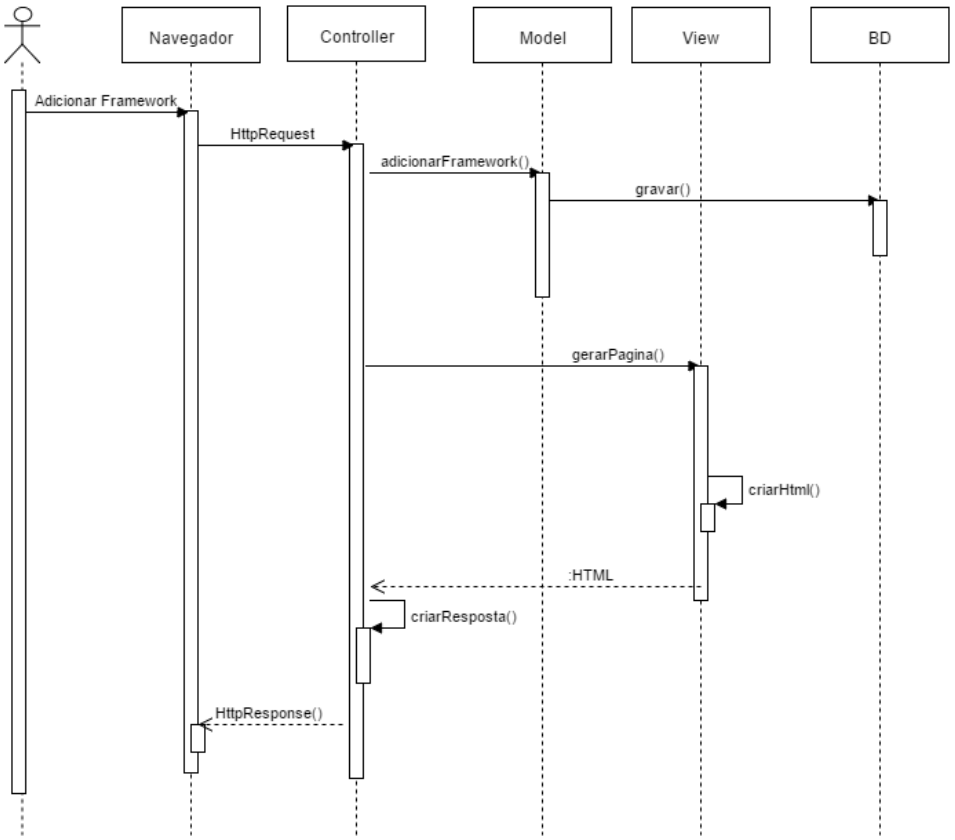

Figura 3 - Diagrama Sequencia MVC

O diagrama de sequência representado na Figura 3 mostra a relação entre os vários componentes deste padrão que são os seguintes:

- Model - Este componente corresponde aos dados e às regras que são aplicados a esses mesmos dados, por exemplo, um correio eletrónico e a data de aniversário têm que ter um certo formato. Os dados serão sempre transmitidos da mesma maneira, independentemente da forma como eles serão apresentados.
- View - Este componente tem a capacidade de fazer a representação dos dados num formato específico. Normalmente o controller é que decide qual a View que vai representar aqueles dados.
- Controller - O controller é o componente que consegue gerir todos os pedidos feitos pelo utilizador, sejam estes pedidos HTTP ou quando o utilizador clica nos elementos interativos da View. A sua principal função é chamar e coordenar os objetos e recursos necessários para efetuar a tarefa pedida.

2.5.3 MVVM

O padrão Model-View-ViewModel [15] é um padrão derivado do MVC, que foi desenvolvido pela Microsoft e que é bastante utilizado nas plataformas .NET e SilverLight.

Tem como objetivo a separação dos componentes e simplificar a programação dos eventos do utilizador.

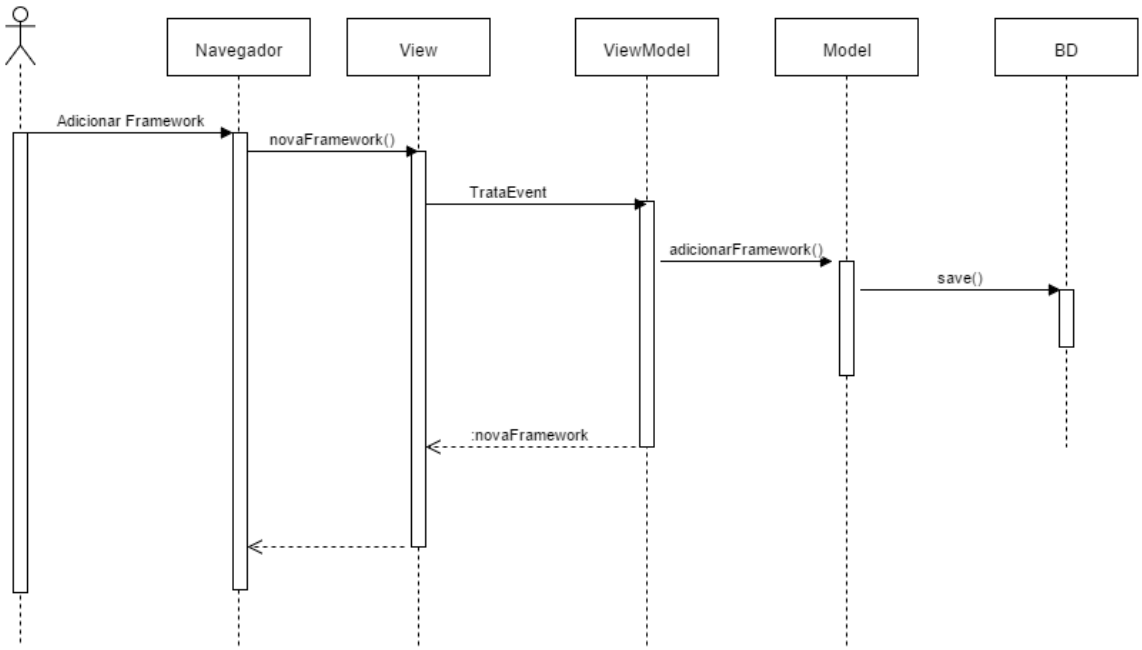

Figura 4 - Diagrama Sequencia MVVM

O diagrama de sequência representado na Figura 4 mostra a relação entre os vários componentes deste padrão que são os seguintes:

- View - Representa a parte de interface com o utilizador e normalmente corresponde a um ficheiro do tipo XAML, que é renderizado de forma adequada pela plataforma para HTML.
- ViewModel - Este constituinte é uma abstração da View que expõe todas as propriedades e comandos da mesma e tem como objetivo interrelacionar as comunicações entre os outros 2 componentes (View e Model), com vista a fazer um *bind* entre os dados do Model e a View. O *bind* entre estes dois componentes, normalmente escrito em linguagem de marcação, permite libertar trabalho de escrever código para sincronizar a View e o Model ao desenvolvedor. Isto irá permitir que quando exista uma alteração nos dados essa mesma alteração seja refletida na View.
- Model - Este elemento normalmente representa o estado da informação e pode ser utilizado também como uma camada de acesso à base dados.

2.6 Inversão do Controlo

Na execução normal, o desenvolvedor tem um conjunto de funções que pode chamar, e que normalmente estão organizadas em classes. Quando uma função é chamada faz algum trabalho e retorna o controlo à função que chamou.

A inversão de controlo, como o próprio nome indica, faz o inverso. Isto é, ao invés de o desenvolvedor ter sempre o controlo de execução, ele delega algumas funcionalidades a terceiros. Portanto, permite criar uma camada de abstração entre as bibliotecas usadas pela frameworks e o código que vai se executado [16].

Existe alguma confusão entre o significado de inversão de controlo e *dependency injection*, pois estes têm características comuns. No entanto, o *dependency injection* é um estilo da inversão de controlo [16].

O desenvolvedor precisa inserir o seu código na framework, quer por subclasse ou ligando as suas próprias classes. O código da framework chama então o seu código (Figura 5).

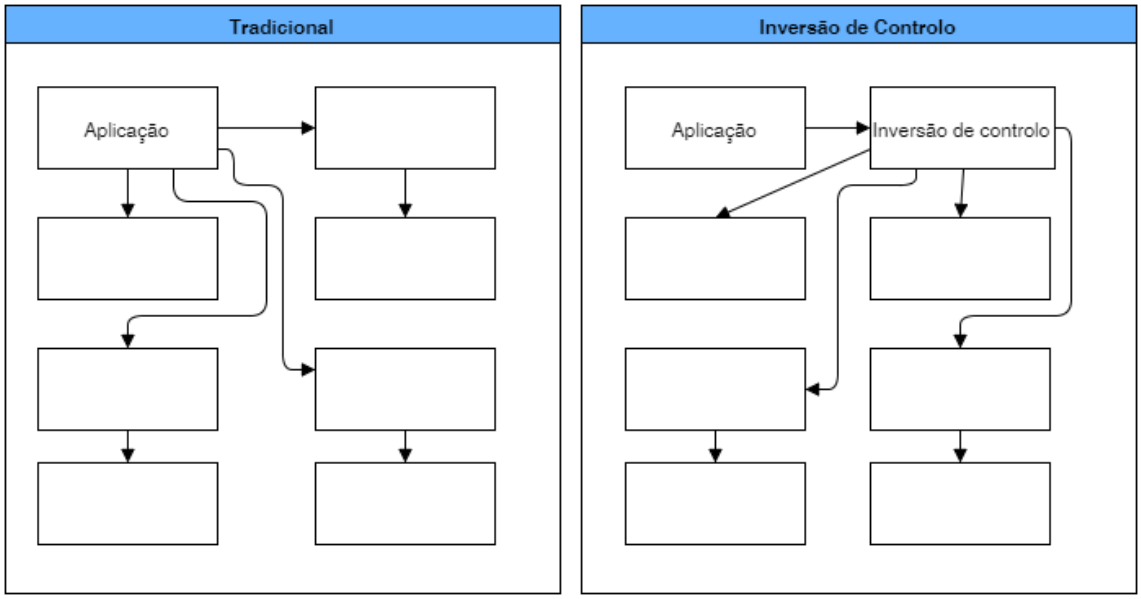

Figura 5 - Inversão de Controlo

Na Figura 5 constam dois exemplos correspondentes à forma como desenvolver *software*, uma é referente à forma tradicional, na qual a aplicação chama todos os componentes externos. A segunda refere-se à inversão de controlo, na qual o código da aplicação é inserido na framework e a este cabe a responsabilidade de chamar os componentes externos.

2.7 Biblioteca vs. Framework

Antes de fazer o levantamento e caracterização das várias frameworks, é necessário caraterizar e diferenciar bibliotecas e plataformas.

Um dos principais aspetos que diferencia uma framework de uma biblioteca é o conceito "inversão de controlo".

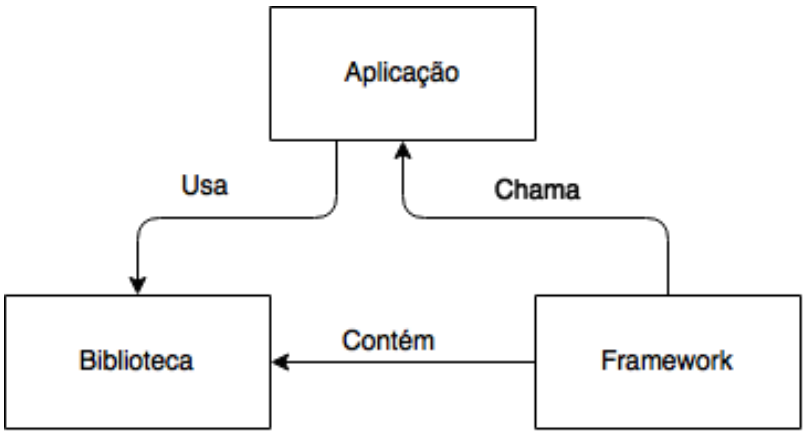

Figura 6 - Biblioteca vs. Framework

A Figura 6 representa como se associam as bibliotecas, aplicações e as frameworks.
Verifica-se assim que uma framework pode conter várias bibliotecas, e chama o código da aplicação. Por sua vez, uma aplicação pode usar bibliotecas, mesmo que estas não pertençam à framework, para atingir um objetivo.

2.7.1 Biblioteca

Uma biblioteca é um repositório de funcionalidades a que o desenvolvedor tem acesso para desenvolver a aplicação., também permite ao desenvolvedor que o mesmo código tenha a mesma performance e compatibilidade entre os vários navegadores de internet.
Um bom exemplo de uma biblioteca é o JQuery, com ela podemos facilmente manipular o HTML, fazer pedidos AJAX e assim aumentar a interatividade de uma página.
A biblioteca permite aplicar inúmeras funcionalidades de uma forma mais simples, permitindo ao desenvolvedor "mandar" a biblioteca executar alguma coisa, havendo sempre o retorno pretendido ("*call and return*"). Trata-se portanto de uma biblioteca, pois o desenvolvedor tem sempre controlo sobre o fluxo.

2.7.2 Framework

Por contraponto, uma framework impõe um conjunto de padrões de desenvolvimento (boas práticas), obrigando o desenvolvedor a segui-los para extrair todo o poder da framework.
De facto, as frameworks adotam uma abordagem "Tu não mandas, nós é que chamamos por ti" [16]. Isto é, a framework está desenvolvida para correr de uma certa maneira e dentro dessa framework existe espaço onde o desenvolvedor pode trabalhar o seu código, e a plataforma irá chamá-lo quando ocorre a situação específica.
Ao utilizá-los o desenvolvedor fica obrigado a ter um código estruturado, fácil de entender e flexível para eventuais alterações que podem ocorrer no futuro. Estes padrões são muitas vezes agnósticos à linguagem de implementação, sendo aplicáveis em várias linguagens de programação.

Para uma melhor compreensão deste conceito será usado o seguinte exemplo:
Imaginemos uma aplicação que consiste num repositório de frameworks e bibliotecas JavaScript. Na página inicial será apresentada a lista de frameworks e bibliotecas caraterizada segundo diversas dimensões (e.g. cliente, servidor ou isomórficos). Nesta página constarão também uma caixa de pesquisa (Figura 7).
Ao selecionar uma plataforma abre-se uma página com informação mais detalhada, como as suas características e todas as informações relativas a esta.

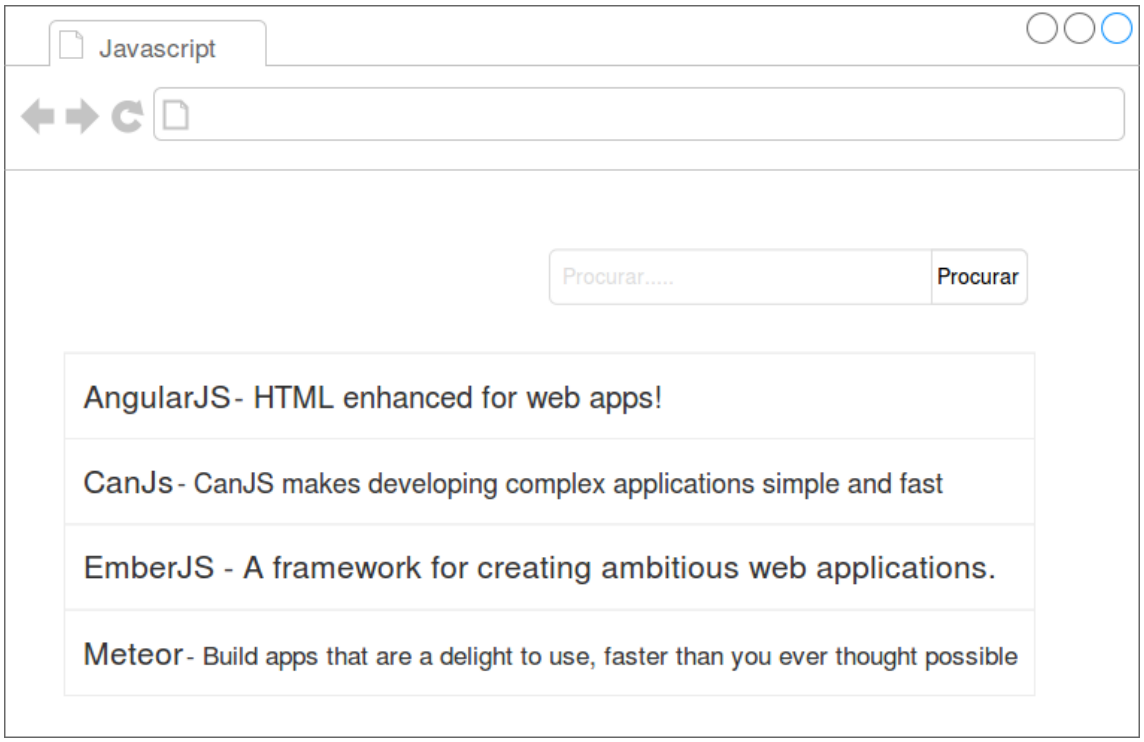

Figura 7 - *Mokup* exemplo

Agora pretende-se dar ao utilizador a possibilidade de guardar o estado da lista em qualquer altura, dá-se assim a possibilidade ao utilizador de ao guardar o *link*, sejam guardadas em simultâneo as ações que permitem voltar ao estado da informação que se estava a ver naquele momento. Isto só é possível se se fizer algum tipo de alteração no URL.
Para isto, é necessário desenvolver código que permita que ao aceder ao *link* específico, a informação esteja no mesmo estado que o guardado, como se pode verificar na Figura 8.

```
angular.module("App_routing", ["ngResource", "App_tagfilter"],
  function($routeProvider) {
    var params = {
    controller: "FrameworkListController",
    templateUrl:"/app/routing/framework_list.html"
    };
  $routeProvider.
    when("/", params).
    when("/filter/:tag", params);
  }
)
```

Figura 8 - Exemplo Routing AngularJS

Este tipo de funcionalidade é muito fácil de desenvolver usando frameworks, porque estas estão preparadas para tal.

Em suma, uma biblioteca ou já suporta algo que o desenvolvedor quer ou caso contrário permite que o desenvolvedor o faça por ele ou com ajuda de terceiros. Por sua vez uma framework ou suporta o que o desenvolvedor pretende ou caso contrário, não permite que o desenvolvedor implemente o que pretende. Por esta razão a escolha da plataforma é algo muito importante para o desenvolvimento de aplicações web.

2.8 Plataformas do lado do Cliente em JavaScript

Nos últimos anos muitas empresas têm apostado em JavaScript para conseguir dar uma melhor experiência ao utilizador em aplicações *Web* [17].

Apesar de se ter tornado *standard*, o JavaScript não estava entre as primeiras escolhas de um desenvolvedor, não só pelos problemas mencionados na secção 2.4 (Evolução do JavaScript), mas também por existirem outras linguagens que permitiam desenvolver software que fornece uma melhor experiência ao utilizador, como flash [18] ou SilverLight [19]. Para além disto, haviam mais desenvolvedores para estas duas linguagens, pois elas também eram utilizadas fora do conceito *Web*.

No entanto, essas linguagens (e.g. Flash) foram descontinuadas ou deixaram de ter o mesmo sucesso e relevância quando o HTML5 foi proposto [20] [21].

Podemos então dizer que o HTML5 foi o catalisador e impulsionador do desenvolvimento de plataformas JavaScript.

As plataformas JavaScript podem ser caraterizadas de acordo com a plataforma onde é executado o *script*:

1. Lado do cliente – Plataformas que correm no lado do cliente para dar uma melhor experiência de utilização. Por exemplo, AngularJS, CanJS, ReactJS,
2. Lado do servidor – Plataformas que correm no lado do servidor e normalmente consistem em responder a pedidos feitos pelo cliente. Por exemplo, ExpressJS [22], NodeJs [23].

3. Isomórficas – Conceito de plataformas que correm nos dois lados. Permitem a reutilização de código em ambas as partes, por exemplo, Meteor [24] ou Rendr [25]. Este tipo de plataformas vai ser desenvolvido com maior pormenor mais à frente neste estudo.

Este trabalho tem como objetivo apresentar as características e todos os conceitos por detrás das plataformas do lado do cliente, sendo também abordado o conceito de plataformas isomórficas.

2.9 Plataformas Isomórficas

Plataformas isomórficas permitem desenvolver aplicações que podem correr no lado cliente e no lado do servidor e são consideradas a próxima etapa na evolução de plataformas JavaScript.

O conceito isomórfico vem do grego "isos" que significa "igual" e "morph" que significa "forma". Basicamente são plataformas que têm a mesma forma em diferentes contextos, neste caso correspondendo ao lado do cliente e ao lado do servidor.

Estas novas plataformas trazem vantagens relativamente às plataformas tradicionais.

A Figura 9 representa de uma forma geral o funcionamento de este tipo de plataformas. A aplicação encontra-se entre o servidor e o cliente e pode ser executada nos dois ambientes. A API é o local onde se encontra a informação que é usada pela aplicação.

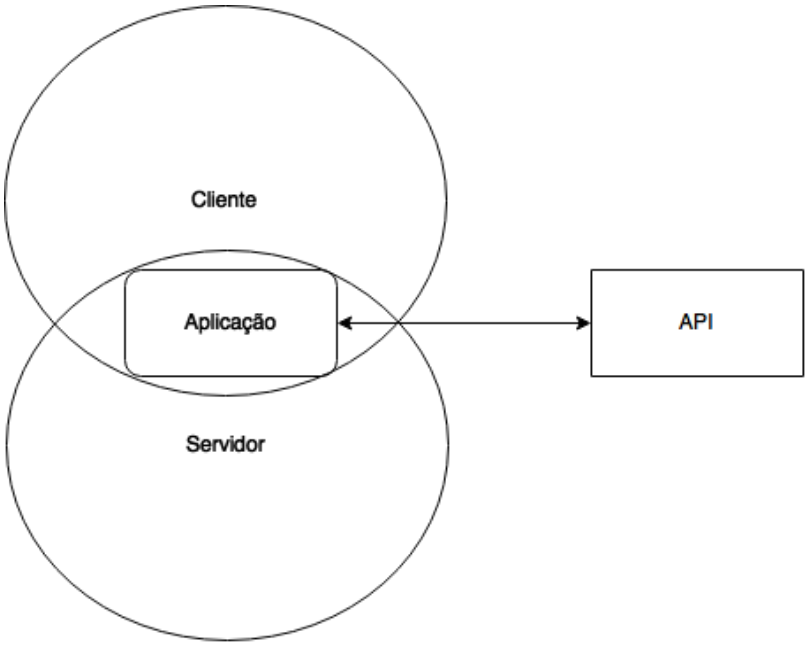

Figura 9- Funcionamento de frameworks isomórficas

A possibilidade de ter a mesma plataforma no lado do cliente ou do servidor permite:

- Simplificar o processo de aprendizagem ao desenvolvedor, visto que possibilita que este se foque apenas numa linguagem de programação;

- Aumentar o desempenho, pois a velocidade de execução destas plataformas é maior ao nível do servidor [26].
- Template Engines

O grande avanço das plataformas obriga a que as bibliotecas estejam cada mais sobre pressão para evoluir e conseguir dar resposta aos avanços que estão a ser dados no desenvolvimento web.

Template Engines são bibliotecas escritas em JavaScript que permitem combinar templates com dados normalmente provenientes do servidor (Figura 10), sendo o seu resultado a criação de páginas *web* (HTML) ou parte delas [27].

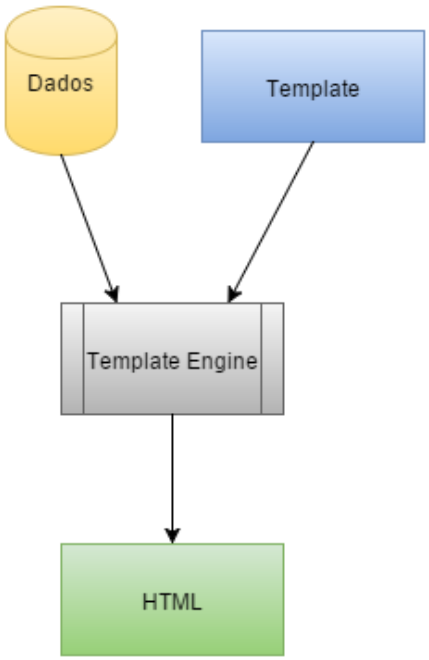

Figura 10 - Funcionamento Template Engine

A estrutura das *templates*, é composta por código HTML e *tags* específicas da biblioteca.
Embora sejam relativamente recentes no JavaScript, este tipo de bibliotecas já existe há vários anos noutras linguagens de programação.
As plataformas, ou grande parte delas, usam padrões de arquitetura por exemplo o MVC ou derivados como o MVVM. Dentro destes padrões as Template Engine têm um importante papel no componente da View (V) porque permitem uniformizar todo HTML e a geração do mesmo.
Neste momento existem vários Templates Engine (e.g. Mustache [28] e Handlebars.js [29]) e como qualquer outra biblioteca, têm as suas vantagens e desvantagens.

Para melhor explicar o funcionamento e o comportamento destas bibliotecas, volta-se a recorrer ao exemplo da lista de plataformas que se está a criar. Imagine-se que será necessária uma

atualização regular da mesma. Surge assim a necessidade de se refrescar (atualizar) a página sempre que existam atualizações aos dados.

Para evitar fazer uma atualização total da página constantemente, utilizam-se as bibliotecas, que visam que se faça apenas a atualização da lista, poupando tempo ao desenvolvedor.

Usando o *mokup* da lista de plataformas representadas na Figura 11.

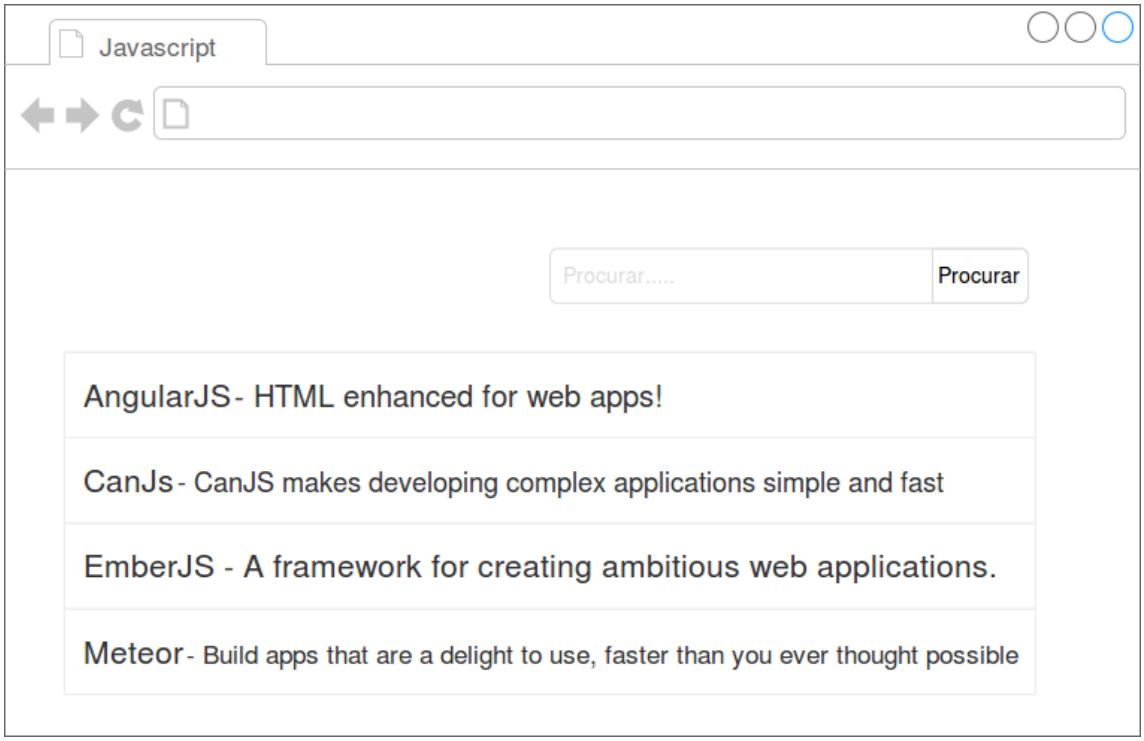

Figura 11 - Layout Framework

Verifica-se que o *layout* para cada framework é sempre igual: consiste num título e uma pequena descrição da mesma (Figura 12).

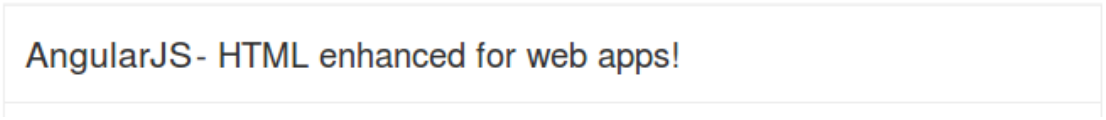

Figura 12 - Layout Framework

Este vai ser o *layout* que vai ser usado sempre que se queira atualizar a lista. Em termos de programação em AngularJS, será algo semelhante à Figura 13.

```
<li data-item="{{name}}" class="item">
  <div class="listing">
    <h3><a href="#">{{name}}</a></h3>
    <p>{{desc}}</p>
  </div>
</li>
```

Figura 13 - Código Template

Para atualizar a lista apenas são necessários os dados que normalmente são provenientes do servidor. Portanto fazendo um pedido AJAX recebe-se a lista de plataformas deste género.

Depois de obter os dados e o template, pode-se gerar a lista e refrescar a página.

2.9.1 Vantagens

As vantagens destas bibliotecas são inúmeras, nomeadamente:

- Reutilização de código,
- A diminuição do tamanho das mensagens feitas ao servidor, devido ao facto de ir apenas buscar dados e nunca receber código HTML, permitindo distribuir o trabalho de gerar partes da página *web* para lado do cliente melhorando a performance do servidor,
- Melhorar a utilização do *website* para o utilizador,
- Permitir a separação de assuntos durante o desenvolvimento, com pouco esforço, que *webdesigners* aprendam a desenvolver os *templates* já formatados nas estruturas da biblioteca e assim facilitar o trabalho ao desenvolvedor, dando a possibilidade de trabalhar em simultâneo.

2.9.2 Desvantagens

Uma das grandes desvantagens das bibliotecas template engines é a complexidade da aprendizagem relativa a todo o funcionamento e estrutura da biblioteca.

Por isso a curva de aprendizagem destas bibliotecas costuma ser alta.

2.10 Benefícios de uso de plataformas JavaScript

Surge a necessidade de se usar uma plataforma JavaScript nas situações seguintes:

- Desenvolvimento de aplicações de uma única página (cf. 3.1),
- Aplicações isomórficas (cf. 3.2),
- Para novas funcionalidades em *websites* já contruídos com vista a melhorar a experiência de utilização ou tentar reduzir o número de pedidos feitos ao servidor (cf. 3.3),

O Gmail, Airbnb e o Linkedin são exemplos de aplicações desenvolvidas com base em plataformas JavaScript.
Os benefícios do uso de plataformas JavaScript no desenvolvimento de aplicações *web* são os seguintes:

- Reduzir a quantidade de código repetido (DRY) [30] ,
- Permitir que o desenvolvedor crie código mais limpo.
- Vantagens na experiencia de utilização para o utilizador da aplicação

Conclui-se que estas plataformas melhoram a experiência do utilizador, ao mesmo tempo que libertam tempo ao desenvolvedor.

3 Cenários de desenvolvimento web

Este capítulo descreve três cenários de desenvolvimento *web* que se consideram representativos e abrangentes do tipo de aplicações web:

- Single Page Application;
- Aplicações isomórficas;
- Widgets.

A descrição e caraterização destes cenários visa proporcionar o contexto de avaliação das frameworks e bibliotecas analisadas.

3.1 Single Page Application

Single page application é um conceito de desenvolvimento de um *website* com uma única página, com o objetivo de tornar a experiência do utilizador mais agradável e mais próxima possível do uso duma aplicação a correr nativamente no dispositivo.
Todo o material necessário para a execução deste tipo de *website* é retirado do servidor no carregamento inicial da página ou é dinamicamente retirado ao longo da execução da aplicação dependendo da interação do utilizador.
A vantagem deste tipo de aplicações é a inexistência da necessidade de fazer recarregamento das páginas ou de alterar a página para apresentar nova informação. As ligações feitas ao servidor para importar informação são feitas sem o utilizador saber.
Além destas vantagens, existem outras como a clara separação de interesses entre o cliente e o servidor e a tentativa de prevenir a necessidade de se partilhar muita lógica entre os dois, visto que normalmente são escritos em linguagens diferentes.
Ao longo dos anos estas aplicações foram concretizadas com diferentes técnicas, no início começou-se com plugins instalados no navegador de internet que ficaram obsoletos, passando

pela utilização do AJAX e neste momento pode-se dizer que a maioria das SPA (single page application) utiliza *websockets* [31].
Quando os navegadores de internet foram concebidos, os seus desenvolvedores não tinham pensado no conceito de SPA, como tal este cenário causou inúmeras dificuldades. O HTML5 foi uma grande ajuda para consolidar este conceito nas páginas *web*.
Este tipo de cenários são normalmente utilizados quando é necessário desenvolver o lado do cliente de raiz com inúmeras funcionalidades sem grandes complicações. Em alguns casos será necessário alterar o lado do servidor para suportar uma interface em REST e JSON para conseguir integrar melhor com a plataforma.

Este conceito de SPA é bastante similar ao conceito Single Document Interface que é muito popular em aplicações nativas para computador.

3.2 Isomórfico

Quando as primeiras plataformas JavaScript começaram a surgir e os desenvolvedores perceberam as suas vantagens, iniciaram a criação de grandes aplicações *web* no navegador de internet. Um exemplo clássico de uma aplicação *web* é o Gmail.
Devido ao excessivo uso destas plataformas, estas nem sempre eram a solução ideal para aplicações *web*, aparecendo assim as suas primeiras deficiências [32].
Entre estas deficiências encontramos algumas que são consideradas mais preocupantes. Por exemplo, se toda aplicação for gerada no lado do cliente os "*crawlers*" não conseguirão cumprir o seu objetivo. Em termos de performance também há algumas melhorias a fazer porque o utilizador tem que esperar algum tempo mais até conseguir carregar toda a página, sendo a experiência de uma página em branco desagradável. Para além disto, observaram-se ainda problemas de manutibilidade do *software* desenvolvido.
De facto, apesar da separação de "interesses", adotada com o objetivo de aumentar o nível de manutenção do *software*, a lógica da aplicação tinha que ser duplicada no lado do cliente como no lado do servidor, e normalmente em linguagens diferentes, o que levantava dificuldades e esforço repetido de desenvolvimento e de manutenção (nomeadamente de consistência entre os dois componentes).
O conceito de plataformas isomórficas permitiu que houvesse uma aplicação no lado do cliente para lhe proporcionar uma melhor experiência e em simultâneo conseguir assegurar a performance e SEO (*crawlers*) no servidor.
Este conceito trouxe então a possibilidade de ter o lado do cliente em JavaScript e o lado do servidor também em JavaScript. Esta nova realidade permite, de uma forma fácil, que nenhum código da lógica seja repetido e que os Models possam ser usados tanto num lado como no

outro, além disso possibilita que o desenvolvedor tenha que conhecer apenas uma linguagem e assim não ter que dividir o seu estudo em duas linguagens.

3.3 Widgets

Os dois últimos casos de uso têm semelhanças, visto que têm como objetivo desenvolver uma página *web* de raiz.

Contudo o mercado já tem inúmeras páginas *web* bem desenvolvidas ao longo dos anos, e como tal, não existe a necessidade de reconstruir o *website* sempre que seja necessário desenvolver novas funcionalidades para o mesmo. Surgiu assim o conceito de Widgets.

Widgets são funcionalidades criadas numa página *web* já existente.

Podem existir diferentes níveis de wigets, por exemplo, quando estamos a desenvolver uma nova funcionalidade para integrar numa localização da página, podemos considerá-la um widget pequeno. No entanto existem widgets quase tão grandes como aplicações de uma página única.

Normalmente para este tipo de casos os desenvolvedores preferem escolher plataformas mais pequenas e com apenas algumas funcionalidades, por serem mais leves e simples de utilizar.

Em suma, este caso compila todas as alterações e novas funcionalidades que sejam necessárias adicionar a *websites* já construídos e sem deteriorar a performance e qualidade do mesmo.

4 Classificação

Neste capítulo constará um levantamento e caracterização das várias frameworks pesquisadas neste estudo.

Estas características são de elevada importância para melhor se poder compreender o funcionamento e aplicabilidade das frameworks. Por esse motivo serão descritas e explicadas neste capítulo.

Como partes integrantes das frameworks, as plataformas serão também explicadas, com vista a compreender-se como funcionam e quais as suas vantagens.

4.1 Características

As características aqui assinaladas são observadas principalmente em plataformas no lado do cliente, por esse motivo, haverão alguns conceitos que não irão ser mencionados, por exemplo os pedidos AJAX e manipulação do DOM.

4.1.1 Two way binding

Esta característica possibilita a criação de uma ligação entre o Model e a View, de tal forma que uma alteração nos dados dum componente seja automaticamente refletida no outro componente. Por outras palavras, sempre que exista alguma alteração no Model, essa mesma alteração será automaticamente refletida na View ou vice-versa.

Isto permite libertar o desenvolvedor de uma enorme quantidade de trabalho, evitando que o mesmo esteja sempre a escrever código para atualizar a View, ou o Model.

Com o uso de bibliotecas do tipo Template Engine este componente tornou-se mais fácil e prático de implementar, permitindo organizar todo o processo de atualização de uma View.

Por exemplo, sempre que a informação é atualizada, a plataforma consegue utilizar a biblioteca de Template Engine e reconstruir parte da View com a nova informação (que foi atualizada).

O mesmo acontece ao contrário, quando existem eventos que acontecem na View e alteram o Model, a plataforma tem o trabalho de atualizar a informação no servidor usando a própria API.

A Figura 14 representa como a sincronização entre o Model e a View funciona juntamente com as bibliotecas Template Engines.

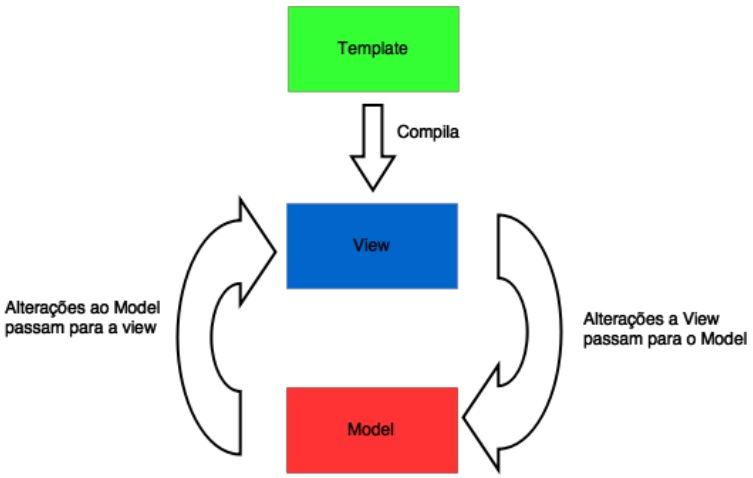

Figura 14 - Two way Binding

4.1.2 Routing

Um dos maiores conceitos por detrás das plataformas JavaScript do lado do cliente é a criação de aplicações de apenas uma página onde somente partes dessa mesma página é que vão sendo alteradas.

Com a alteração do URL do *website* é possível representar o conjunto de ações, e com isto permitir ao utilizador avançar ou retroceder no navegador sem fazer o recarregamento da página. Este conjunto de ações encontra-se na parte final do URL, parte essa que normalmente começa com #.

Outra vantagem desta característica é a possibilidade de permitir ao utilizador guardar o conjunto de ações nos favoritos quando este grava o URL.

Por exemplo, pretende-se filtrar a lista de plataformas (exemplo mencionado ao longo do estudo) para ver apenas as plataformas do lado do cliente.

Com o esquema da Figura 15 torna-se mais simples compreender o funcionamento do Routing.

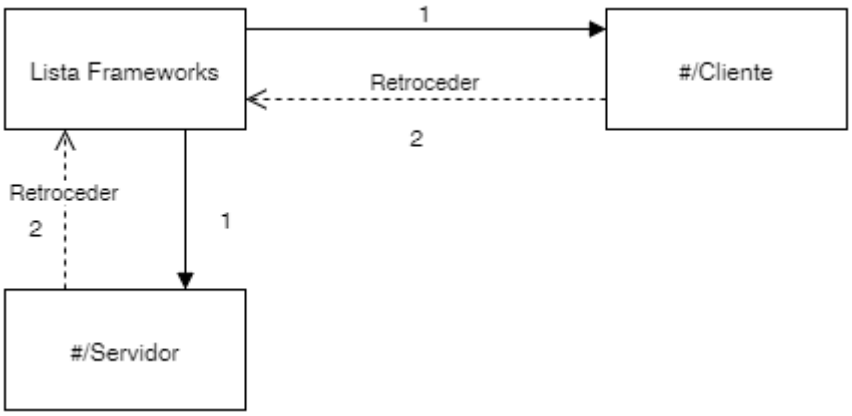

Figura 15 - Routing

Pode usar-se o routing para filtrar a lista de plataformas, para realizar essa filtragem basta juntar-se à base do URL a expressão "#/Cliente", que automaticamente fará com que a lista seja filtrada com plataformas que operam somente no lado do cliente.

Esta alteração no URL do navegador de internet permite ao utilizador guardar as ações que levaram a este estado da informação nos seus favoritos e caso queira voltar para a lista original apenas terá que retroceder no próprio navegador de internet. O mesmo procedimento acontece caso se queira ver plataformas do lado do servidor.

4.1.3 Template Engines

As Template Engines permitem que o desenvolvedor possa adicionar à plataforma com a qual está a trabalhar a Template Engine mais adequada ao seu trabalho.
Normalmente as plataformas já utilizam bibliotecas conhecidas (Third-Party) e, portanto, esta característica é comum. Ainda assim, existem algumas plataformas que utilizam bibliotecas desenvolvidas internamente na plataforma, o que se pode tornar num problema, pois o desenvolvedor pode ter que aprender como esta funciona e quais as suas limitações.

4.1.4 Ferramentas de Teste

Testar o código produzido por processos automáticos é algo cada vez mais comum nos processos de desenvolvimento de *software* [33]. Sem essa possibilidade é complicado ter-se uma consciência sobre a qualidade do mesmo e identificar de onde podem surgir problemas.
Uma das maneiras mais simples e eficazes para testar uma aplicação é através do uso de testes unitários, permitindo testar vários módulos da aplicação separadamente e assim ser mais fácil identificar os problemas ocorridos.
Esta funcionalidade também possibilita o acesso a diversas funções específicas para fazer "*debug*" ao código.

4.1.5 Padrão de Arquitetura

Os padrões arquiteturais adotados pela framework condicionam o desenvolvimento, manutibilidade, suportabilidade, etc. Além disso, o padrão arquitetural afeta a produtividade do desenvolvedor.

4.1.6 Linguagem e Formatos do Servidor

Tendo as aplicações *web* a parte do lado do cliente e do lado do servidor, muito da lógica é desenvolvida na parte do servidor, nas quais são usadas variadas linguagens de programação, como PHP, .NET, JAVA e Python.

Esta caraterística pode ser importante caso suporte o formato das mensagens provenientes do servidor.

4.1.7 Curva de aprendizagem

A curva de aprendizagem [34] permite aferir o esforço necessário para entender conceitos e tecnologias necessárias para tirar partido da plataforma ou combinação de plataformas, o que condiciona a sua escolha.

Para além de arquiteturas diferentes potenciarem produtividade, há que considerar ainda as preferências ou conhecimento prévio do desenvolvedor que potencie a aprendizagem. De facto, se a plataforma usar conceitos já conhecidos e utilizados noutras plataformas, ou se tiver um conjunto de tutoriais e exemplos conhecidos e fornecidos ao público, então pode-se considerar que a plataforma facilita a aprendizagem. Caso a plataforma tenha poucos tutoriais ou estes não sejam muito explícitos ou usarem conceitos novos, conclui-se que a aprendizagem é longa pois o desenvolvedor poderá vir a ter que gastar mais tempo a entender a plataforma do que a desenvolver a aplicação *Web*.

4.1.8 Tamanho

O tamanho da plataforma pode parecer a característica menos importante aqui assinalada mas tem implicações em vários fatores:

- Quando existe uma fraca ligação à internet. Por exemplo, quando são desenvolvidas aplicações *Web* para países pouco desenvolvidos, em desenvolvimento ou onde toda a estrutura de rede de comunicação é fraca, o tamanho da aplicação é uma variável muito importante.
- Quando a aplicação *web* se destina a plataformas *mobile* e o uso dos dados móveis seja reduzido com vista a reduzir o tráfego de dados;
- Quando a performance é um fator muito valorizado ou crucial de uma aplicação *Web*.

Se o tamanho da plataforma for reduzido, mais rápido será feito o *download* do mesmo e por sua vez mais rápido será feito o carregamento da página.

Os valores dos tamanhos comparados são de ficheiros de produção, estes mesmos ficheiros encontram-se minificados e comprimidos em gzip.

4.1.9 Comunidade

Esta característica tem como objetivo principal avaliar o tamanho da comunidade que cada framework tem e quanto ativa é essa comunidade. Esta característica permite aos desenvolvedores ter a noção da quantidade de pessoas que programam sobre a framework e a possibilidade de apoio para os problemas que possam vir a surgir.

4.1.10 Suporte do Navegador de Internet Mínimo

O Suporte Navegador de Internet Mínimo é a caraterística que indica a versão mínima dos principais navegadores que existem no mercado.
Esta característica é sempre avaliada em todas plataformas e bibliotecas, pois permite saber qual os navegadores mínimos compatíveis com a plataforma. Quanto menores forem as versões nos vários navegadores, mais abrangente essa plataforma é.
Esta informação permite aferir da aceitabilidade da plataforma, o que pode ser relevante nomeadamente nos casos de desenvolvimento de aplicações para certas organizações, países, etc. com particularidades de dispositivos e navegadores usados.

4.1.11 Isomórfica

Esta característica permite classificar a plataforma segundo a capacidade nativa de correr no cliente (normalmente no navegador) e no servidor.
Caso se esteja a desenvolver uma aplicação de raiz ou se existir num futuro próximo a possibilidade de passar o trabalho realizado do lado do cliente para o lado do servidor, esta caraterística pode ser relevante.

4.1.12 Suporte comunitário

As melhores formas para verificar a grandeza da comunidade é acedendo a dados estatísticos dos principais *websites* especializados para controlo de versões e de ajuda. Neste caso os seguintes sites foram usados como referência:

- GitHub
- StackOverFlow

A avaliação do suporte comunitário vem das informações retiradas destas plataformas ao longo da pesquisa.
Na Tabela 1 podem observar-se os resultados obtidos na pesquisa efetuada.

Tabela 1 - Plataformas e Comunidade

	GitHub Watchers	StackOverFlow Followers	StackOverFlow Questions	Resultado
AngularJS	3500	14 300	104500	Muito Boa
EmberJs	1116	2000	15872	Boa
CanJs	129	54	188	Fraca

Reactjs	1777	1487	3711	Boa
Meteor	1587	1803	13309	Boa
Rendr	246	5	13	Muito Fraca
Backbone	1624	4400	18254	Muito boa
Knockout	602	2800	14700	Boa

4.2 Frameworks

A escolha das plataformas a usar neste estudo baseou-se em dois objetivos:

- Apresentar diferentes plataformas com ideais consequentemente diferentes
- Plataformas populares no mercado e algumas outras não tão conhecidas.

Neste último caso além de classificar plataformas como CanJs, KnoukoutJS que representam uma alternativa as plataformas mais conhecidas, e.g. AngularJS [35], EmberJS [36], também se abordaram algumas plataformas como Meteor e Rendr, que representam um novo conceito de plataformas e bibliotecas e que nos últimos anos têm vindo a angariar cada vez mais desenvolvedores e fãs por todo mundo [32].

Os tópicos seguintes apresentam as plataformas estudadas neste relatório de uma forma sucinta.

4.2.1 AngularJS

AngularJS é uma plataforma focada no lado do cliente e que funciona bastante bem com servidores que oferecem uma interface em REST/JSON.

Esta plataforma foi desenvolvida usando essencialmente os padrões arquiteturais Model-View-Controller e "Dependency Injection".

O AngularJS for inicialmente desenvolvido em 2009 por Miško Hevery e Adam Abrons como um *software* por trás de um serviço de armazenamento JSON online, que teria preço estimado por megabyte, para aplicações pré-construídas (*easy-to-make*) para as empresas [37].

Este empreendimento foi disponibilizado em "GetAngular.com", e teve alguns aderentes, antes de os dois desenvolvedores decidirem abandonar a ideia comercial e distribuir o AngularJS como uma plataforma *open-source*.

Abrons deixou o projeto, mas Hevery, que trabalha na Google, continuou o seu desenvolvimento e manteve a plataforma em conjunto com alguns colegas do Google: Igor Minár e Vojta Jína.

Antes aprofundar todas as características desta plataforma, convém verificar-se a Figura 16.

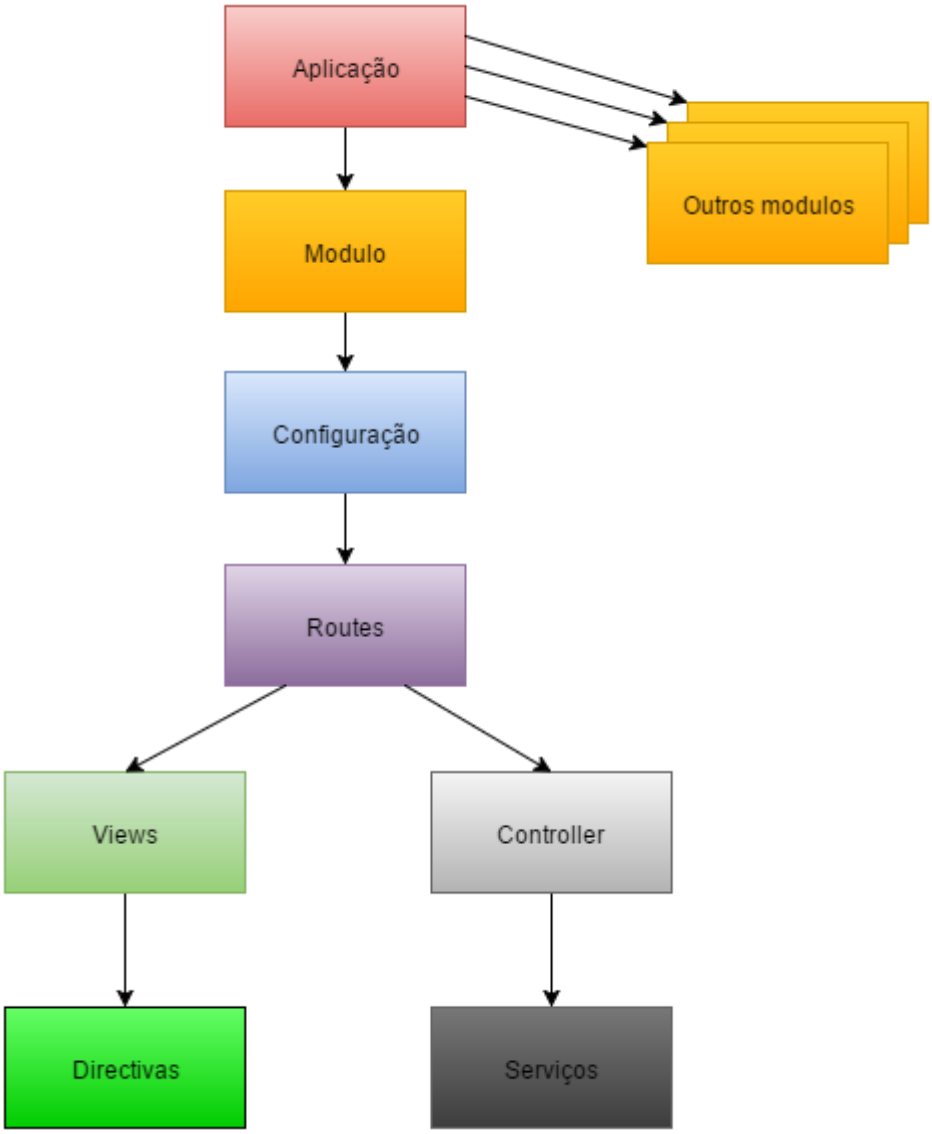

Figura 16 - Funcionamento AngularJS

O AngularJS é uma plataforma modular que contém vários componentes, como e.g.:

- Serviços: Mecanismo de sincronismo entre o Model e o servidor.
- Controllers: Prepara o Model para a View e recebe as *callbacks* da View.
- Directivas: As directivas HTML são especiais atributos nos elementos HTML que permitem manipular o DOM e renderizar valores dinâmicos,
- Router: Local onde é possível definir os URLs para a navegação da aplicação

Uma das mais importantes caraterísticas e vantagens desta plataforma é o uso "dependency injection" como forma de adotar a Inversão do Controlo. Tal permite, de uma forma fácil, interligar partes de código e módulos, permitindo ao desenvolvedor manter um código com baixo acoplamento.

Além disso o AngularJS permite ter acesso a inúmeros serviços e funcionalidades nativos, contando sempre com uma comunidade muito grande. O facto de ser promovido e mantido pela Google contribui para a perceção de ter um grande e bom suporte (comunitário) [37].

4.2.2 CanJS

CanJS é uma plataforma JavaScript que tem como principal objetivo reduzir a quantidade de código que o desenvolvedor tem que criar, permitindo que quando no código deste surja uma pequena alteração, a plataforma se encarregue de fazer os *updates* necessários [35].
O objetivo é que quando é feita uma alteração ao Model, o CanJS com ajuda do padrão de desenvolvimento Observer consiga executar todas as alterações necessárias automaticamente.
O CanJS permite ao desenvolvedor criar uma aplicação *web* utilizando o padrão de arquitetura Model-View-Controller. Com esta plataforma, o código mantém-se modular com uma clara separação de tarefas [35].
Quando o desenvolvedor usa CanJS, consegue usar objetos que notificam *listeners* quando a informação é alterada. Isto permite que os componentes fiquem separados, porque faz com que eles se comunicam por eventos, em vez de referências diretas uns com os outros.
Usando estes conceitos o CanJS consegue de uma forma simples implementar grandes funcionalidades como:

- Two bing binding;
- Routing;
- Views parciais.

Outra grande vantagem desta plataforma é o seu tamanho relativo, apesar de necessitar de uma biblioteca como JQuery, ou Mootools por exemplo.
Existem diversos componentes que fazem com que esta plataforma funcione com podemos ver na Figura 17.

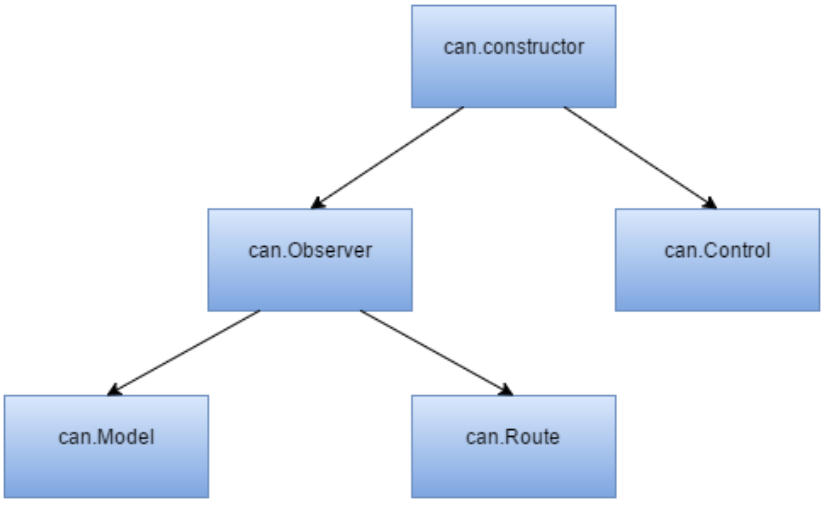

Figura 17 - **Funcionamento CanJS**

4.2.3 Meteor

Meteor [24] permite desenvolver de uma forma fácil e para várias plataformas muito rapidamente.

Por ser uma plataforma isomórfica, permite que o desenvolvedor não tenha que aprender duas linguagens diferentes. Como o desenvolvedor pode programar o lado do cliente e o lado do servidor, também é uma plataforma desenvolvida para dar a melhor experiência ao utilizador, tendo como características o *Two Way Binding* e uma biblioteca nativa de templates.

A única falha que esta plataforma poderia conter é a falta de um sistema de routing, mas este problema já foi resolvido antes da versão 1.0 com o desenvolvimento do package Iron Router.

Outra grande vantagem desta plataforma é a integração no lado cliente do mongoDB, que permite replicar informação que está no lado do servidor. Esta vantagem traz algumas funcionalidades especiais como:

- "*Live queries":* Possibilidade de receber *updates* sobre a *query* que foi feita anteriormente;
- *"Tracker-ready":* Sempre que haja uma alteração na base dados a plataforma executa algum código;
- *"Change tracking":* Possibilidade de guardar a informação que vai sendo alterada permitindo que, caso seja necessário, seja possível reverter essas alterações.

Estas funcionalidades só são aplicadas com o uso do protocolo Distributed Data [38] e com o padrão publish-subscriber [24].

Neste momento o Meteor é desenvolvido pela Meteor Development group.

4.2.4 EmberJS

EmberJS [36] é uma plataforma JavaScript para o lado do cliente, desenvolvida a pensar no padrão Model-View-Controller (MVC) e tem como principal objetivo ajudar os desenvolvedores que pretendem construir aplicações *web* escaláveis de uma única página.

Foi construída sobre o JQuery, que permite esconder as várias inconsistências existentes entre os navegadores de internet e adicionar um conjunto de funcionalidades e bibliotecas como Handlerbars para a renderização de templates HTML.

Para além disso, o EmberJS pode ser considerado um conjunto de regras ou convenções que já foram provadas como robustas e com elevados níveis de performance.

Um dos seus principais pilares é a "Convenção sobre configuração", podendo isto ser uma vantagem ou desvantagem dependendo da pessoa que está a desenvolver.

O conceito "Convenção sobre configuração" consiste em evitar que o desenvolvedor se repita, impondo-lhe um conjunto de convenções a seguir, tornando assim o código mais simples e mais fácil de ler [39].

Se por exemplo, o desenvolvedor gostar de seguir a convenção de como certo comportamento deve ser realizado, então o EmberJS será uma boa escolha. No entanto o desenvolvedor pode sentir a necessidade de realizar alterações e por isso mexer em todos os componentes para tentar melhorar a sua aplicação ou para a controlar com a maior precisão possível, então EmberJS pode não ser uma boa escolha.

Olhando para a documentação do EmberJS verifica-se que com poucas linhas de código a plataforma consegue executar várias funcionalidades devido ao conceito anteriormente explicado. Isto permite que a plataforma já esteja preparada para ter um comportamento normal em muitas situações.

Em suma, o EmberJS é uma plataforma com diversas funcionalidades e vantagens como qualquer outra plataforma, tendo características como Two Way Binding ou sistema de Routing.

4.2.5 React

React [40] é uma biblioteca que visa criar e desenvolver interfaces de utilizadores para projetos de uma página única e foi desenvolvida em colaboração entre o Facebook e o Instagram.

Neste momento é mantida por estas duas grandes empresas e por desenvolvedores à volta do mundo e tem sido usada para outras aplicações criadas por outras empresas, não só pelo Facebook e Instagram mas também empresas como Yahoo, Airbnb, Sony entre outras.

Por não ser uma plataforma, não tem a quantidade de componentes que existem em plataformas como Ember ou AngularJs.

Esta biblioteca tem como principal objetivo ajudar os desenvolvedores a criar componentes de interfaces de utilizadores que permitam que a informação se possa atualizar ao longo do tempo.

As suas características são diferentes das plataformas como AngularJs.

No AngularJs existem características de Two Way Binding e no React existe a *One Way Binding*.

Um dos maiores conceitos deste projeto é o Virtual DOM. Pode-se pensar no virtual DOM como uma página virtual que se consegue modificar e alterar a página real. Este conceito mostra ser extremamente eficiente porque a página final só iria receber as alterações depois de um algoritmo ter verificado quais iriam ser essas alterações.

Neste momento existem diversos desenvolvedores que utilizam o React para gerar uma página estática inicialmente no servidor e de seguida utilizam a mesma biblioteca para concretizar as interações do utilizador e as atualizações necessárias no lado do cliente.

4.2.6 Rendr

Rendr é uma biblioteca que permite utilizar aplicações Backbone no lado do cliente e no lado do servidor. Isto permite que a sua aplicação *web* usufrua de todas as vantagens do Backbone nos dois lados [25].

Um dos objetivos desta biblioteca é permitir desenvolver melhores aplicações *Web*, não sendo necessário desenvolver duas aplicações em linguagens diferentes para conseguir responder a todos os desafios.

Apesar de tudo o Rendr não visa ser uma biblioteca com todas as capacidades, a base do Backbone, apenas tem o necessário para começar a desenvolver, permitindo ao desenvolvedor utilizar no futuro as melhores ferramentas da forma que achar mais adequada para a sua aplicação.

4.2.7 KnockoutJS

Knockout é uma plataforma JavaScript que implementa o padrão Model-View-ViewModel com o uso de *templates*.

Tem como principal característica o Two Way Binding. Esta característica é suportada pelo conceito de *declarative binding*, que permite que caso hajam alterações no HTML desenvolvido, este se consiga adaptar, caso os elementos continuem a ter a mesma hierarquia [41].

A plataforma já contém uma biblioteca de *templates* nativa, mas caso o desenvolvedor queira, pode utilizar qualquer biblioteca que exista no mercado.

O seu tamanho, 20KB usando compressão HTTP e o suporte a grande parte dos navegadores de internet até mesmo os mais antigos, tornam a plataforma ainda mais apetecível aos desenvolvedores.

Knockout é desenvolvido e mantido como um projeto *Open source* por Steve Sanderson, um colaborador da Microsoft, contudo o autor já veio indicar que Knouckout não faz parte da Microsoft.

4.2.8 Backbone

Backbone [17] é uma pequena biblioteca JavaScript que permite que o desenvolvedor siga um conjunto de princípios e com isso melhore a estrutura da aplicação no lado do cliente.

Esta biblioteca pode ser usada para criar aplicações *web* de uma única página. Estas aplicações permitem que o navegador de internet consiga reagir às alterações no lado do cliente sem ter que se refrescar a página a partir do servidor.

O Backbone já existe há vários anos e contém uma grande comunidade que vai adicionando plug-ins e extensões, tornado possível adicionar uma nova funcionalidade a uma aplicação apenas adicionando bibliotecas externas.

Backbone tem como principal objetivo oferecer ao desenvolvedor um conjunto de funcionalidades que ajudam a manipular a informação em vez de reinventar os objetos JavaScript. O seu tamanho é uma das grandes vantagens para desenvolver aplicações para *smartphones*.

Devido ao backbone ser uma biblioteca, funciona bastante bem com outras bibliotecas, permitindo adicionar widgets a aplicações já desenvolvidas.

5 Comparação

A comparação das plataformas e bibliotecas tem como objetivo mostrar a relação que existe entre estas plataformas, as suas características e os cenários em que estas plataformas podem ser usadas.

Na figura 18 podemos ver a relação existente entre plataformas, características e os cenários.

Figura 18 - Comparação Plataformas/Características/Cenários

Cada plataforma tem nativamente um conjunto de características que pode ser alargado recorrendo a caraterísticas e funcionalidades de bibliotecas externas que permitam completar ainda mais a plataforma.

Por fim, mas não menos importante, verifica-se também uma relação entre as características e cenários. Normalmente uma plataforma necessita de um conjunto de características específicas que permitam satisfazer um cenário. Portanto, existe uma relação entre as características que uma plataforma tem de ter e as que um cenário necessita.

Uma plataforma pode ser usada num ou mais cenários, ou noutra perspetiva, uma plataforma deve ser usada de forma a potenciar as suas caraterísticas para a adoção nos cenários que delas necessita.

5.1 Cenários e Características

Na tabela 2 captura-se as caraterísticas das plataformas necessárias ao desenvolvimento de cada tipo de aplicação (cenário). Através da pesquisa e análise de conteúdos realizada ao longo do projeto para cada cenário, determinaram-se quais as características que são obrigatórias (++), convenientes (+) ou desnecessárias (-).

Tabela 2 - Cenários e Características

Caraterística / Cenário	Two Way Binding	Routing	Template Engine	Isomorphics	Ferramentas de testes	Tamanho	Suporte mínimo do navegador de internet
SPA	++	++	++	-	+	+	+
Aplicação isomórfica	+	+	+	++	+	+	+
Widgets	+	+	+	-	+	++	+

Numa SPA, é obrigatório que a plataforma tenha características como Two-Way Binding, Routing, Template Engines. Caso estas características não constem na plataforma, o desenvolvimento deste cenário torna-se complexo.

Para uma aplicação *web* isomórfica, o leque de características obrigatórias torna-se reduzido, pois a única característica obrigatória que a plataforma tem que ter é isomorfismo. Caso tal caraterística não seja disponibilizada, então o desenvolvimento deste tipo de aplicação pressupõe o desenvolvimento dessa característica de raiz.

No desenvolvimento de um widget, as características referem-se especificamente ao tipo de widget pretendido. Neste tipo de *software* é consideravelmente difícil decidir quais as características obrigatórias, sendo todas elas aceites, cabendo ao desenvolvedor verificar qual a mais adequada. Por esta razão apenas foi adicionado o critério de obrigatoriedade à característica "tamanho", por motivos de performance.

5.2 Características e Plataformas

Nesta secção as plataformas analisadas no capítulo anterior são descritas segundo as caraterísticas enumeradas anteriormente (Tabela 3)

Tabela 3 - Características e Plataformas

	AngularJS	CanJS	EmberJS	React	Meteor	Rendr	Backbone	Knockout
Two Way Binding	*Digest cycle/ Dirty checking*	*Observable Properties*	*Observable Properties*	-	*Observable Properties*	*Observable Properties*	*Observable Properties*	*Observable Properties*
Routing	*Deep Linking*	*Deep Linking*	*pushState History*	-	-	*pushState History*	*pushState History*	-
Template Engines	Incluida	Externa	Externa	Incluida	Externa	Externa	Externa	Incluida
Isomorphics	Não	Não	Não	Não	Sim	Sim	Não	Não
Ferramentas de Teste	Karma	QUnit	Qunit	Jtest	Laika	Jasmine/ Sinon.JS	Jasmine/ Sinon.JS	Jasmine/ Sinon.JS
Tamanho	40kB	20KB	136KB(Jquery + Handlebars)	35 KB	200KB	198KB	13KB(Underscore + json2)	44KB (jQuery)

Suporte comunitário	Muito Boa	Fraca	Boa	Boa	Boa	Muito Fraca	Muito boa	Boa
Suporte Mínimo do Navegador de Internet	IE: 9 Chrome:22 Firefox:16 Safari:5 Android:4 IOS:6.1	IE: 6 Chrome:22 Firefox:16 Safari:5.1 Android:2.3 IOS:6.1	IE: 8 Chrome:22 Firefox: 16 Safari: 5.1 Android:2.3 IOS:6.1	IE: 9 Chrome:31 Firefox: 38 Safari: 5.1 Android:2.3 IOS:6.1	IE: 8 Chrome:22 Firefox:16 Safari: 4 Android:2.3 IOS:6.1	IE: 10 Chrome:31 Firefox:38 Safari:8 Android:4.3 IOS:8.1	IE: 10 Chrome:31 Firefox:38 Safari:8 Android:4.3 IOS:8.1	IE: 6 Chrome:5 Firefox:4 Safari:5 Android:2.3 IOS:6.1
Padrão de Arquitectura	MVC	MVC	MVC	MVVM	MVVM	MVC	MVC	MVVM
Linguagem e Formatos no Servidor	REST / (XML ou JSON)	REST / (XML ou JSON)	REST / (XML ou JSON)	REST / JSON	REST / (XML ou JSON)	REST / JSON	REST / (XML ou JSON)	REST / JSON
Curva de Aprendizagem	Alta	Baixa	Alta	Intermédia/ Alta	Intermédia	Baixa	Baixa	Intermédia

Algumas das plataformas são mais densas em características do que outras, mas isso não as torna melhores do que outras. Apenas mostra que foram criadas com diferentes propósitos.
Apesar das características serem genéricas, a forma como cada característica é executada pode variar, cabendo ao desenvolvedor a escolha da forma que considerar mais correta ou que julgar que irá satisfazer melhor as necessidades da sua aplicação.
Por exemplo, relativamente à caraterística Two Way Binding, a Tabela 3 mostra que o AngularJS é a única plataforma a ter um comportamento diferente com a utilização de um "*digest cycle"*, enquanto as outras plataformas usam "*observable properties*".
Digest Cycle é um ciclo que verifica a sincronização entre o Model e a View e ocorre num certo intervalo de tempo. *Observable properties* consiste na observação de propriedades ao longo da execução da aplicação.
O *Routing* é também uma das características que são comparadas na tabela, esta caraterística mostra que o AngularJS e o CanJS usam *deep linking,* enquanto as restantes frameworks usam *pushstate.history.*
Deep linking consiste em alterar o URL para conter informações específicas sobre um conteúdo para melhorar a pesquisa e indexação [42]. Enquanto *pushState History* é responsável por alterar o URL do navegador sem iniciar nenhum pedido ao servidor [43].
Por outro lado, podemos observar que a característica Template Engines pode ter o seu próprio motor em cada uma das plataformas, ou utilizar até um dos vários Template Engine que existem atualmente no mercado.
Analisando a comparação efetuada relativamente ao suporte mínimo do navegador de internet, verifica-se quais são as versões mínimas dos navegadores de internet mais conhecidos no mercado que cada plataforma suporta.
Salvaguarde-se que estes valores foram retirados de documentos oficiais relativos a cada plataforma, *websites* especializados em funcionalidades HTML5 e apenas correspondem à utilização nativa ou à sua utilização em conjunto com bibliotecas dependentes. Assim, na maioria dos casos é possível utilizar as plataformas para versões anteriores às assinaladas, com a utilização de bibliotecas que permitem maior compatibilidade.

5.3 Plataformas e Cenários

Com base nas duas tabelas anteriores, é possível sugerir a adequação de plataformas a cada um dos cenários considerados (Tabela 4). Uma plataforma pode estar preparada (++), ter possibilidade (+) ou não estar preparada (-).
Como se pode observar na Tabela 4, as plataformas mais adequadas para a construção de uma SPA são as seguintes: AngularJS, EmberJS e CanJS.

Por outro lado, as aplicações isomórficas são casos particulares porque são poucas as plataformas que suportam isomorfismo, nomeadamente: Meteor e Rendr.
No desenvolvimento de um widget existem algumas plataformas que estão à partida excluídas deste cenário, porque o conceito geral dessas plataformas é construir um ecossistema que é o oposto ao que este quer retratar. I.e., têm caraterísticas a mais do que as necessárias, o que as torna inconvenientes. Neste cenário, o objetivo é integrar sistemas já construídos e para isso já existem algumas plataformas como Backbone e Knockout que têm como objetivo desenvolver aplicações para este tipo de casos.

Tabela 4 - Plataformas e Cenários

Plataformas/Use Case	SPA	Isomorphics	Widgets
AngularIS	++	-	-
CanJs	++	-	-
EmberJs	++	-	-
ReactJs	+	-	+
MeteorJs	+	++	-
Rendr	+	++	+
Knockout	+	+	++
BackBone	+	+	++

5.4 Apreciação final

Os cenários anteriormente abordados permitiram avaliar a aplicabilidade de cada plataforma em relação a um conjunto de características importantes para os cenários.
Esta avaliação ao nível dos cenários permite abranger ao máximo as necessidades do mercado, favorecendo uma melhor avaliação de cada uma das plataformas, porque estas também foram desenvolvidas a pensar em cenários diferentes.
Se bem que os resultados anteriores apresentem factos que podem ajudar na escolha da plataforma a adotar em três tipos de aplicações, trata-se apesar do resultado de uma análise e sistematização nas quais o autor incluiu uma certa dose de subjetividade.

6 Avaliação

Existem inúmeras plataformas e bibliotecas no mercado, tornando a escolha duma plataforma em detrimento de outra uma tarefa difícil e em muitos casos subjetiva.

Devido a esta dificuldade, decidiu-se levar a cabo um inquérito a desenvolvedores de aplicações *web* seguindo os mesmos princípios que foram adotados no capítulo anterior.

Os resultados do inquérito permitirão perceber quais são as características consideradas importantes para cada cenário, possibilitando perceber-se quais as reais necessidades do mercado e assim sendo, as plataformas que melhor se adequam.

6.1 Inquérito

O inquérito é constituído por duas páginas, na primeira página (Figura 19) constam um conjunto de questões de escolha múltipla relacionadas com o desenvolvimento de aplicações *web* para cada um dos vários cenários propostos neste projeto.

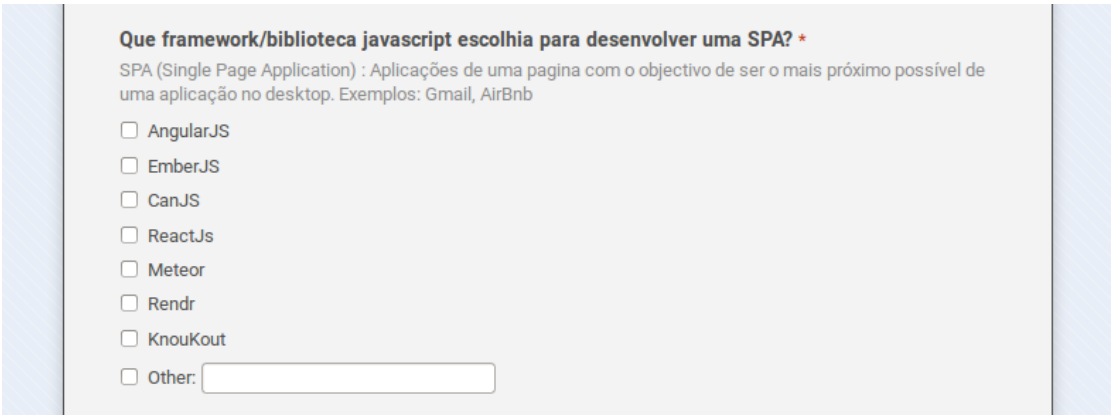

Figura 19 - Exemplo de pergunta de plataforma

A segunda página (Figura 20) diz respeito a um conjunto de perguntas que visam a avaliação das características funcionais necessárias para cada cenário.

Esta parte do inquérito tem como objetivo a identificação das características mais relevantes para cada cenário, e serão avaliadas nos seguintes níveis de conhecimento:

- Não sabe
- Não é relevante,
- Pouco relevante
- Relevante
- Muito relevante.

Figura 20 - Exemplo de pergunta características

Na parte final do inquérito (Figura 21) recorreu-se à mesma metodologia utilizada nas questões supracitadas, com vista a avaliar a relevância das características não funcionais das plataformas: curva de aprendizagem e a comunidade de suporte.

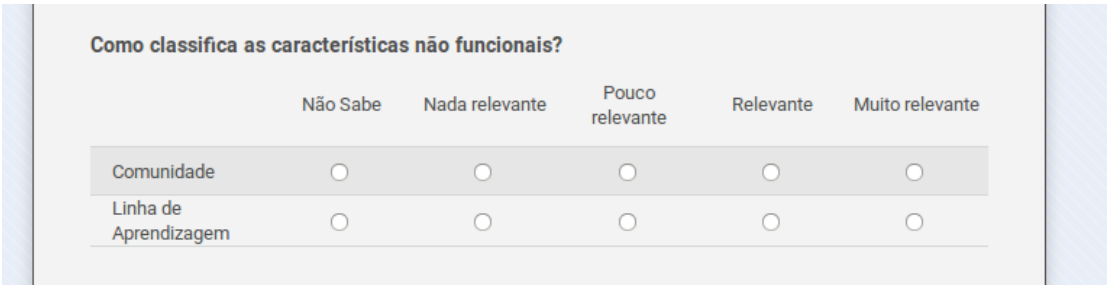

Figura 21 - Pergunta características não funcionais

6.2 Análise de resultados

O inquérito foi enviado a 50 desenvolvedores na área, dos quais 27 responderam, dando um total de 54% da amostragem.

Os resultados obtidos através do inquérito realizado aos desenvolvedores na área focou-se em três cenários, divididos nas tabelas abaixo.

6.2.1 Cenário SPA

Para o cenário SPA os inquiridos escolheram as seguintes características como mais relevantes.

Na Tabela 5 constam as percentagens relacionadas com o número de relevância de cada uma das características.

Tabela 5 - Análise SPA

	Não sabe	Não é relevante	Pouco Relevante	Relevante	Muito Relevante
Two Way binding	7.4%	3.7%	7.4%	44.4%	29.6%
Routing	8%	0%	4%	52%	36%
Template Engine	4%	0%	24%	32%	40%
Ferramentas de Teste	0%	4%	12%	64%	20%
Tamanho	4%	4%	40%	20%	32%
Padrão de Arquitectura	0%	4%	16%	44%	36%
Suporte Mínimo do Navegador de Internet	0%	0%	16%	36%	48%
Linguagem e Formatos no servidor	8%	4%	28%	44%	16%

Analisando a tabela verifica-se que a maioria dos inquiridos escolheu como relevantes e muito relevantes as quatro primeiras características apresentadas acima, contudo o suporte mínimo do navegador de internet também tem uma grande percentagem de adesão.

Assim, as características definidas como mais relevantes para este cenário são as seguintes: Two Way Binding, Touting, Template Engine e ferramentas de teste.

Através da análise dos dados conclui-se que a característica "tamanho" para este caso de uso não é vista como importante, sendo que 40% dos inquiridos a definem como pouco relevante.

Para este cenário o gráfico abaixo (Figura 22) representa quais as plataformas escolhidas pelos inquiridos para desenvolver uma aplicação *Web*.

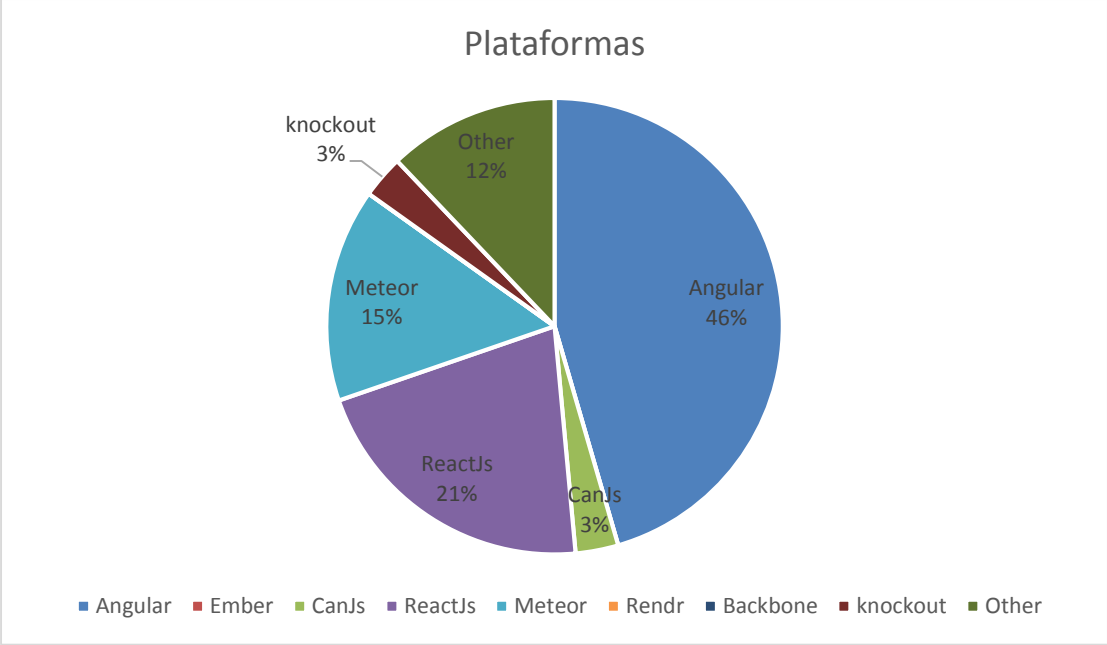

Figura 22 - Plataformas SPA

6.2.2 Cenário Isomórfico

Relativamente ao Cenário Isomórfico, também se definiu quais as características consideradas mais relevantes, sendo essa informação posteriormente convertida para percentagens (Tabela 6).

A Tabela 6 relativa ao Cenário Isomórfico revela que, mais uma vez, as quatro primeiras características são consideradas como relevantes e muito relevantes.

As características selecionadas foram as seguintes: Two Way Binding, Routing, Template Engine e ferramentas de teste.

Tabela 6 - Análise Isomorphics

	Não sabe	Não é relevante	Pouco Relevante	Relevante	Muito Relevante
Two Way binding	11.1%	3.7%	7.4%	22.2%	48.1%
Routing	16%	0%	0%	48%	36%
Template Engine	12%	0%	8%	56%	24%
Ferramentas de Teste	12%	4%	8%	56%	20%
Tamanho	12%	4%	28%	28%	28%

Padrão de Arquitectura	12%	4%	12%	40%	32%
Suporte mínimo do Navegador de Internet	12%	0%	16%	36%	36%
Linguagem e Formatos no S.	16%	0%	24%	44%	16%

O gráfico abaixo (Figura 23) indica qual a plataforma escolhida pelos inquiridos, caso fossem desenvolver uma aplicação neste cenário.

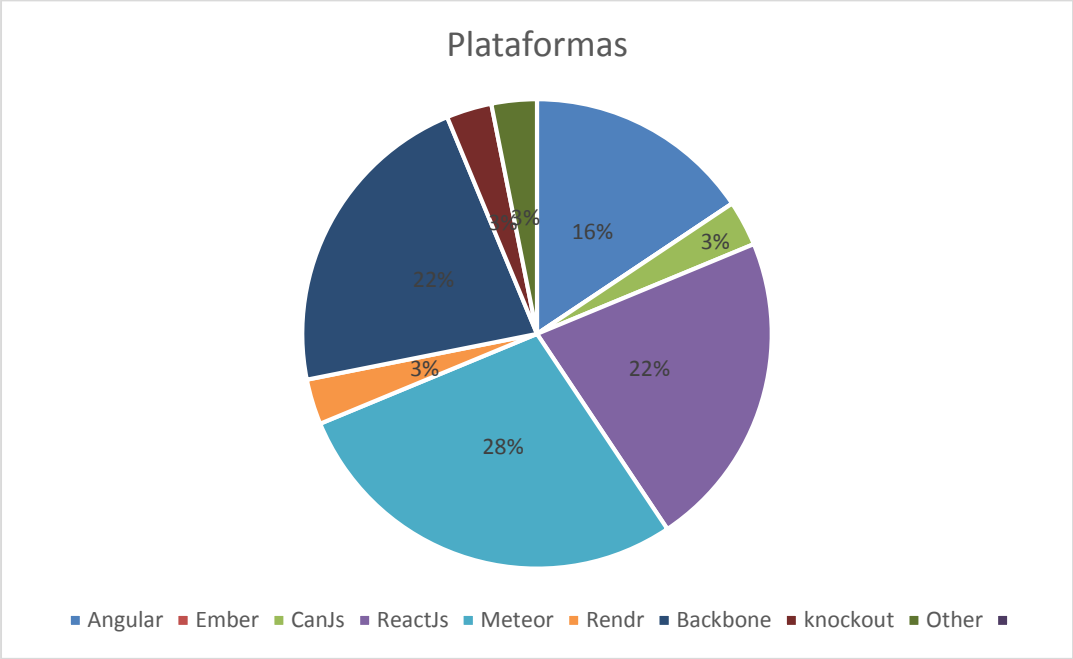

Figura 23 - Plataformas Isomórficas

6.2.3 Widget

No que diz respeito ao cenário de widget, utilizou-se o mesmo procedimento que foi utilizado nos outros cenários e os resultados foram os seguintes (Tabela 7):

Ao contrário dos resultados obtidos em relação aos cenários anteriores, para este cenário os inquiridos revelaram dar mais enfoque a características anteriormente vistas como pouco relevantes, o que é o caso do "tamanho da plataforma".

Também foram destacadas como relevantes as Template Engine, as ferramentas de teste e o Suporte mínimo do navegador de internet, sendo consideradas as principais características para satisfazer este cenário.

A característica Padrão de arquitetura também teve uma alteração acentuada relativamente à sua relevância, conquistando 56% dos votos.

Tabela 7 - Análise Widget

	Não sabe	Não é relevante	Pouco Relevante	Relevante	Muito Relevante
Two Way binding	11.1%	11.1%	29.6%	18.5%	22.2%
Routing	12%	16%	16%	32%	24%
Template Engine	4%	4%	20%	44%	28%
Ferramentas de Teste	4%	4%	24%	44%	24%
Tamanho	0%	4%	28%	24%	44%
Padrão de Arquitectura	0%	8%	16%	56%	20%
Suporte mínimo do navegador de internet	0%	0%	16%	44%	40%
Linguagem e Formatos Back-End	8%	8%	36%	24%	24%

O próximo gráfico (Figura 24) representa qual seria a plataforma escolhida pelos inquiridos, caso tivessem que desenvolver novas funcionalidades para este cenário.

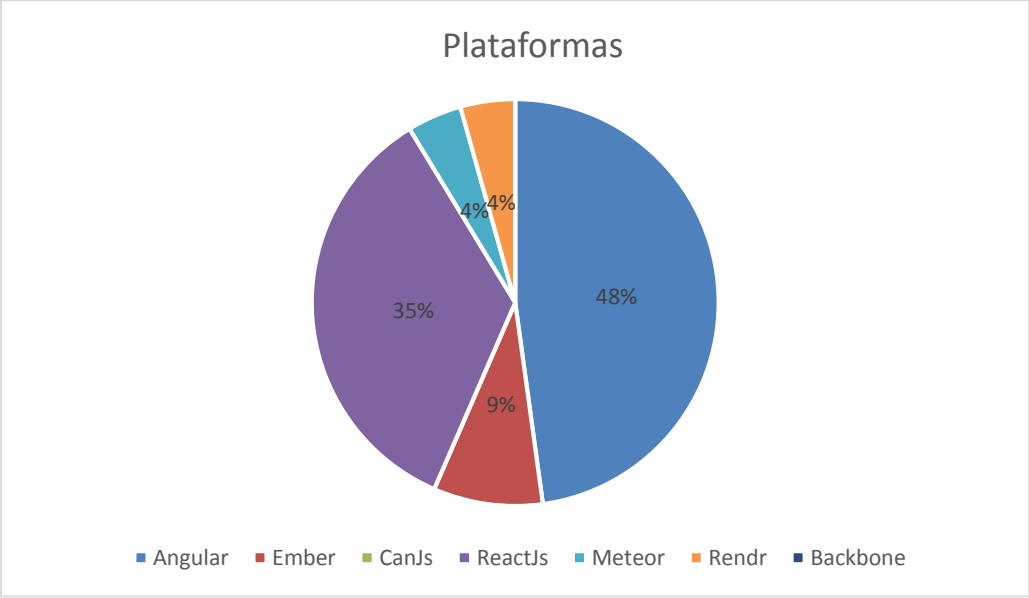

Figura 24 - Plataformas e Widgets

6.2.4 Características não funcionais

Relativamente às características não funcionais, verificou-se que os inquiridos as consideram relevantes ou muito relevantes (Tabela 8).

Tabela 8 - Características não funcionais

	Não sabe	Não é relevante	Pouco relevante	Relevante	Muito Relevante
Linha de Aprendizagem	0%	0%	0%	68%	32%
Comunidade	0%	0%	0%	60%	40%

6.3 Análise Subjectiva

Nesta secção consta uma análise subjetiva do estudo realizado relativamente às plataformas JavaScript.

Os resultados do inquérito realizado mostram uma perspetiva diferente de tudo o que foi abordado ao longo do estudo e da investigação efetuada inicialmente.

Segundo a vária investigação efetivada na área, os cenários de SPA e isomórficos são normalmente ecossistemas e consequentemente as melhores ferramentas para desenvolver estes cenários serão sempre plataformas, isto porque uma plataforma pela sua definição, na maioria dos casos, já é um ecossistema, permitindo dar ao desenvolvedor um conjunto de funcionalidades bem estruturadas e com a máxima facilidade possível.

Dito isto, no cenário de SPA a plataforma mais apropriada seria o AngularJS, porque esta plataforma traz todas as funcionalidades necessárias para este cenário e é a plataforma com mais maturidade referida neste documento, sendo por isso um fator muito importante para escolha desta plataforma.

Para o cenário Isomórfico, Meteor seria a melhor opção visto ser uma plataforma desenvolvida principalmente para este cenário.

O desenvolvimento desta plataforma teve de raiz o princípio de satisfazer este cenário, sendo assim uma plataforma que tem evoluído muito nos últimos tempos e desenvolvendo alguns protocolos como DDP [38] e funcionalidades que poderão ser usadas por outras plataformas, por exemplo, minimongo [44].

Para concluir, para o cenário de widget a melhor opção seria a plataforma Backbone.

Este cenário não é considerado um ecossistema, é apenas um conjunto de novas funcionalidades. Este foi o cenário que mais contrastou entre os resultados da pesquisa realizada ao longo do projeto e os resultados do inquérito.

Usar uma plataforma como AngularJS ou EmberJS nestes cenários iria dificultar o seu desenvolvimento e as suas vantagens não seriam superiores às desvantagens trazidas.

Por esse motivo, as bibliotecas serão possivelmente a melhor opção nesta situação.

Em suma, a escolha desta plataforma será a melhor solução, não só pelo seu tamanho mas também pela sua facilidade em integrar outras bibliotecas, permitindo ter as mesmas vantagens que bibliotecas como Knockout.

7 Conclusão

Neste capítulo constam as conclusões finais do estudo realizado, e um pequeno resumo das ações realizadas para chegar a estas conclusões.

Existem inúmeras plataformas e bibliotecas no mercado, tornando o ato de escolha de uma plataforma em detrimento de outra, uma tarefa quase inconcebível.
Devido a esta dificuldade ao escolher, considerou-se neste estudo, que a melhor solução seria avaliá-las em relação aos vários cenários já previamente estabelecidos.
Os cenários foram avaliados em relação à sua aplicabilidade em relação a um conjunto de características importantes para os mesmos, o que permitiu abranger ao máximo as necessidades do mercado e favorecendo uma melhor avaliação de cada uma das plataformas, porque estas também foram desenvolvidas a pensar em cenários diferentes.

7.1 SPA (Single Page Application)

Considerou-se a partir do ponto 6.2 Análise de resultados, que as características mais relevantes para este cenário são:

- Two way binding,
- Routing,
- Template Engine,
- Suporte mínimo do navegador de internet,
- Ferramentas de teste.

Cruzando os dados da tabela de comparação entre as plataformas e características consideradas essenciais pelos inquiridos, para este cenário, chegou-se a conclusão de que as frameworks mais compatíveis com este cenário são as seguintes:

- AngularJS,
- EmberJS
- CanJS.

No entanto, no inquérito realizado, também se procurou saber qual a plataforma que os inquiridos consideravam mais útil para o cenário.
Os resultados apresentados mostram que 55.6% dos inquiridos escolheu o AngularJS. Em segundo lugar ficou o React, com 25.6% dos votos.
Pode-se assim concluir que a plataforma que mais se adapta a este cenário é o AngularJS.

7.2 Isomorphics

A partir do ponto 6.2 Analise de resultados chegou-se à conclusão que os inquiridos consideram mais relevantes para este cenário as seguintes características:

- Two way binding,
- Routing,
- Template Engine,
- Suporte mínimo do navegador.

Comparando estes resultados com os dados que constam na tabela "Características e Plataformas", conclui-se que as plataformas mais compatíveis com este cenário são o Meteor e o Rendr. Ressalva-se que estas plataformas contém características isomórficas.

Para além de se tentar saber quais as características mais adequadas a este cenário, também se procurou saber qual a plataforma que os inquiridos consideravam mais útil para o cenário.

Os resultados revelam que 33.3% dos inquiridos escolheu o Meteor. As plataformas React e Backbone, obtiveram o segundo lugar com 25.6% de adesão.

Dito isto, podemos considerar que a plataforma mais adequada a este cenário é a Meteor.

Este tipo de aplicações tornou-se possível com o surgimento de diversas plataformas nomeadamente NodeJS, Grunt e Browsify. Embora já não seja novo, este conceito só recentemente ganhou inércia de adoção pelo desenvolvimento de mais plataformas e de grandes projetos sobre essas plataformas.

Contudo, a maturidade deste conceito e das respetivas plataformas é ainda assim provavelmente a maior desvantagem que se pode apontar. De facto, prevê-se que a sua evolução e novos conceitos, abordagens e *standards* tornarão ainda mais simples o desenvolvimento e manutenção do *software*, ao mesmo tempo que permitirão melhorar a experiência do utilizador.

7.3 Widgets

Como referido anteriormente, o ponto 6.2 Análise de resultados revelou quais as características mais relevantes para cada cenário.

Assim sendo, as características consideradas mais importantes para este cenário foram as seguintes:

- Routing,
- Template Engine,
- Tamanho,
- Suporte mínimo do navegador de internet
- Ferramentas de teste.

Através da tabela "Características e Plataformas" verifica-se que as características selecionadas se identificam mais com as seguintes plataformas:

- Backbone
- React

- Knockout

Conclui-se assim que estas são as mais adequadas a este cenário.

O inquérito também visava saber qual a plataforma que os inquiridos consideravam mais útil para este cenário.

O AngularJS ficou em primeiro lugar com 40.7% de adesão, e 29.6% dos inquiridos elegeu React e Backbone como as plataformas mais úteis.

Apesar de a maioria dos inquiridos ter considerado o AngularJS como a melhor plataforma para este cenário, chegou-se à conclusão, através de toda a investigação realizada na área, que esta não é a plataforma mais adequada a este cenário.

Contudo as plataformas que foram escolhidas em segundo lugar, já são plataformas mais consideráveis para este cenário.

Conclui-se que a biblioteca que satisfaz melhor este cenário é o Backbone por conter todas as características consideradas como determinantes anteriormente. Salienta-se o facto de esta plataforma ser a que tem o tamanho mais reduzido, sendo este um fator determinante para este cenário.

7.4 Apreciação final

Uma grande parte dos objetivos propostos inicialmente, foram alcançados neste estudo, nomeadamente a identificação do que o mercado procura com os casos de uso identificados. Para além disto foi possível comparar as várias características das plataformas.

Inicialmente o número de plataformas que iriam ser estudadas neste projeto era mais elevado. Porém, devido à falta ou inexistente informação e falta de maturidade das plataformas, algumas delas não foram incluídas neste projeto, pois não conseguiam concorrer diretamente com as outras plataformas.

7.5 Apreciação final pessoal

A realização deste projeto foi muito gratificante, permitindo-me consolidar e adquirir novos conhecimentos sobre plataformas JavaScript e todos os conceitos associados. Tive a oportunidade de conhecer novas plataformas e conhecer as suas características e as suas vantagens e desvantagens, podendo vir a usar esse conhecimento no meu futuro profissional.

A possibilidade de interligar e comparar as diversas plataformas usando os cenários descritos neste projeto permitiu que a avaliação destas mesmas plataformas fosse mais objetiva e conclusiva. Em resumo, sinto-me satisfeito com os objetivos atingidos com este estudo.

Infelizmente o desenvolvimento e maturidade de grande parte das plataformas ainda é muito fraca, por esse motivo ainda não devem ser consideradas para ambientes de produção, por ainda serem muito instáveis. Uma das melhorias que considero que poderiam trazer valor para

este estudo seria a inclusão de outras plataformas, o que não foi possível pelos motivos mencionados anteriormente.

8 Referências bibliográficas

[1] *JavaScript Cookbook*, 2 edition. Sebastopol, CA: O'Reilly Media, 2015.

[2] A. Pike, "Javascript Framework fatigue," *allenpike*. .

[3] "What Is SEO / Search Engine Optimization?," *Search Engine Land*. [Online]. Available: http://searchengineland.com/guide/what-is-seo. [Accessed: 25-Oct-2015].

[4] *Ajax: The Definitive Guide*, 1 edition. Beijing ; Cambridge MA: O'Reilly Media, 2008.

[5] M. Wiki, "A Short History of JavaScript," *W3*. 27-Jun-2012.

[6] "JavaScript creator ponders past, future," *InfoWorld*, 23-Jun-2008. [Online]. Available: http://www.infoworld.com/article/2653798/application-development/javascript-creator-ponders-past--future.html. [Accessed: 23-Oct-2015].

[7] "Details of the object model," *Mozilla Developer Network*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model. [Accessed: 25-Oct-2015].

[8] *Professional JavaScript for Web Developers*, 3 edition. Indianapolis, IN: Wrox, 2012.

[9] "From STUPID to SOLID Code!" [Online]. Available: http://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/. [Accessed: 24-Oct-2015].

[10] "appjs/appjs," *GitHub*. [Online]. Available: https://github.com/appjs/appjs. [Accessed: 24-Oct-2015].

[11] M. Robert C., *Principle and Patterns*. .

[12] *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3 edition. Upper Saddle River, N.J: Prentice Hall, 2004.

[13] *Head First Design Patterns*, 1st edition. Sebastopol, CA: O'Reilly Media, 2004.

[14] *Patterns of Enterprise Application Architecture*, 1 edition. Boston: Addison-Wesley Professional, 2002.

[15] Microsoft, "MVVM Pattern," *The MVVM Pattern*. 10-Feb-2012.

[16] "bliki: InversionOfControl," *martinfowler.com*. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html. [Accessed: 21-Oct-2015].

[17] *Developing Backbone.js Applications*, 1 edition. Sebastapol, Calif: O'Reilly Media, 2013.

[18] A. Flash, "Flash About." .

[19] Microsoft, "silverlight." .

[20] I. Kantor, "Javascript vs Flash." .

[21] J. Meyer, "JavaScript vs Silverlight," *Blog • Bitovi.com*. .

[22] "strongloop/expressjs.com," *GitHub*. [Online]. Available: https://github.com/strongloop/expressjs.com. [Accessed: 25-Oct-2015].

[23] G. Shipley, "StrongLoop | Getting Started with Node.js for the PHP Developer." .

[24] I. Strack, *Getting Started with Meteor.js JavaScript Framework - Second Edition*, 2 edition. Packt Publishing - ebooks Account, 2015.

[25] "rendrjs/rendr," *GitHub*. [Online]. Available: https://github.com/rendrjs/rendr. [Accessed: 25-Oct-2015].
[26] "Improving performance on twitter.com," *Twitter Blogs*. [Online]. Available: https://blog.twitter.com/2012/improving-performance-on-twittercom. [Accessed: 25-Oct-2015].
[27] *Full Stack Web Development with Backbone.js*, 1 edition. Beijing: O'Reilly Media, 2014.
[28] "mustache/mustache.github.com," *GitHub*. [Online]. Available: https://github.com/mustache/mustache.github.com. [Accessed: 25-Oct-2015].
[29] "wycats/handlebars.js," *GitHub*. [Online]. Available: https://github.com/wycats/handlebars.js. [Accessed: 25-Oct-2015].
[30] D. Thomas, "DRY(Don't repeat yourself)."
[31] C. Missal, "Single Page Apps & Realtime, a Love Story | Chris Missal's Blog." .
[32] "Isomorphic JavaScript: The Future of Web Apps," *Airbnb Engineering*. [Online]. Available: http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/. [Accessed: 25-Oct-2015].
[33] "bliki: UnitTest," *martinfowler.com*. [Online]. Available: http://martinfowler.com/bliki/UnitTest.html. [Accessed: 25-Oct-2015].
[34] "Curva de Aprendizagem." .
[35] *Seven Web Frameworks in Seven Weeks: Adventures in Better Web Apps*, 1 edition. Dallas, TX ; Raleigh, NC: Pragmatic Bookshelf, 2014.
[36] *Building Web Apps with Ember.js*, 1 edition. Beijing ; Sebastopol: O'Reilly Media, 2014.
[37] "GetAngular," 13-Apr-2010. [Online]. Available: http://web.archive.org/web/20100413141437/http://getangular.com/. [Accessed: 25-Oct-2015].
[38] Meteor, "Meteor DDP." [Online]. Available: https://www.meteor.com/ddp. [Accessed: 26-Oct-2015].
[39] "Convention vs. Configuration." [Online]. Available: http://www.dailykos.com/story/2015/09/14/1420744/-Convention-vs-Configuration. [Accessed: 26-Oct-2015].
[40] "A JavaScript library for building user interfaces | React." [Online]. Available: https://facebook.github.io/react/index.html. [Accessed: 25-Oct-2015].
[41] *Knockout.js: Building Dynamic Client-Side Web Applications*, 1 edition. O'Reilly Media, 2015.
[42] "Deep Linking," *W3*. .
[43] "HTML5 History: Clean URLs for Deep-linking Ajax Applications." [Online]. Available: http://www.codemag.com/article/1301091. [Accessed: 27-Oct-2015].
[44] Meteor, "Meteor Mini Databases." [Online]. Available: https://www.meteor.com/mini-databases. [Accessed: 26-Oct-2015].