Danilo Celestino de Castro
Poliana Roberta Schnornberger

CoreQuery
Um Framework para Desenvolvimento de Aplicações Web

Palmas - TO

2017

Danilo Celestino de Castro
Poliana Roberta Schnornberger

CoreQuery
Um Framework para Desenvolvimento de Aplicações Web

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Sistemas para Internet do Instituto Federal do Tocantins - Campus Palmas, como exigência à obtenção do grau de Tecnólogo em Sistemas para Internet.

Instituto Federal de Educação, Ciência e Tecnologia do Tocantins - Campus Palmas
Curso Superior de Tecnologia em Sistemas para Internet

Orientador: Prof. Fagno Alves Fonseca

Palmas - TO
2017

Castro, Danilo Celestino de
Schnornberger, Poliana Roberta

CoreQuery: Um Framework para Desenvolvimento de Aplicações Web / Danilo Celestino de Castro e Poliana Roberta Schnornberger. – Palmas - TO, 2017.

51 f.

Trabalho de Conclusão de Curso (Graduação Tecnológica em Sistemas para Internet) – Instituto Federal de Educação do Tocantins, Campus Palmas, 2017.

Orientador: Prof. Fagno Alves Fonseca

1. Framework. 2. Corequery. 3. Front-End. 4. HTML. 5. Javascript. I. Título

Danilo Celestino de Castro
Poliana Roberta Schnornberger

CoreQuery
Um Framework para Desenvolvimento de Aplicações Web

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Sistemas para Internet do Instituto Federal do Tocantins - Campus Palmas, como exigência à obtenção do grau de Tecnólogo em Sistemas para Internet.

Este Trabalho de Conclusão de Curso foi julgado e aprovado para obtenção do título de Tecnólogo em Sistemas para Internet, no Curso Superior de Tenologia em Sistemas para Internet, do Instituto Federal de Educação, Ciência e Tecnologia do Tocantins. Palmas - TO, ____ de ____________ de 2017.

Prof. Fagno Alves Fonseca, Msc.
Orientador

Prof. Marinaldo Oliveira Santos, Msc.
Avaliador

Prof. Paulo da Silva Paz Neto, Msc.
Avaliador

Prof. Mauro H. Lima de Boni, Msc.
Avaliador

Palmas - TO
2017

Agradecimentos

Agradecemos ao nosso professor orientador Fagno Alves Fonseca pela dedicação e paciência durante a elaboração deste trabalho.

"Eu acredito que é possível pessoas
comuns escolherem ser extraordinárias"
Elon Musk

Resumo

Desde os primórdios da internet, sistemas e aplicações *web* vem cada vez mais se difundindo entre os desenvolvedores. Isto se dá, principalmente ao fato de que esta, ao contrário de aplicações *desktops*, são compatíveis com as mais diversas plataformas e dispositivos, sem demanda de instalação ou configuração. Em aplicações *web* é possível também a elaboração de interfaces gráficas ricas através do uso de uma vasta gama de recursos disponíveis nas linguagens de programação e tecnologias atuais. A popularização das aplicações *web* fez com que desenvolvedores intensificassem esforços na elaboração e aperfeiçoamento de métodos e técnicas, com objetivo de aumentar a produtividade durante o desenvolvimento e otimizar o desempenho das aplicações produzidas. Dentre as soluções encontradas para satisfazer estes objetivos, destaca-se o uso de *frameworks*. Visando contribuir e otimizar a produtividade durante o desenvolvimento *web*, este trabalho propõe a construção de um *framework* que tem como finalidade simplificar o desenvolvimento através da abstração de funcionalidades genéricas utilizadas pelo navegador para executar a atualização do DOM (*Document Object Model*). Os resultados obtidos com esta pesquisa demonstram que o *Framework* CoreQuery fornece módulos e funções que demandam um maior reúso de código na manipulação de elementos na camada de visualiação.

Palavras-chaves: Framework; Corequery; Front-End; HTML; Javascript.

Abstract

Since the beginning of the internet, systems and web applications, is becoming more popular among developers. The main reason is the fact that, unlike desktop applications, Web Applications are compatible with all platforms and devices, without the need for installation or configuration. In web applications it's also possible to build rich graphical interfaces by using a wide range of features available in current programming languages and technologies. The popularization of web applications has made web developers search for methods and techniques aiming to increase production during development and optimize the performance of the applications produced. Among the solutions found for these purposes, we highlight the use of Frameworks. As a goal to contribute and optimize productivity during web development, this work proposes build a framework that aims to simplify the development through the abstraction of generic functionalities used by the browser to perform the Document Object Model (DOM) update. The results achieved with this work demonstrate that the CoreQuery Framework provides modules and functions that demand a larger reuse of code on manipulation of elements in the view layer.

Keywords: Framework. Corequery. Front-End. HTML. Javascript.

Lista de ilustrações

Figura 1 – Exemplo de *template* utilizando Handlebars.js. 19
Figura 2 – Exemplo de *template* utilizando Underscore.js Microtemplates. 19
Figura 3 – Padrão de Arquitetura MVC 21
Figura 4 – Padrão de Arquitetura MVVM 23
Figura 5 – Diferença entre o padrão de arquitetura MVC e MVVM 25
Figura 6 – Exemplo de hierarquia DOM 26
Figura 7 – Representação de interação entre *framework*, biblioteca e código. 28
Figura 8 – Padrão de Arquitetura TMV (TreeModel-View) 30
Figura 9 – Exemplo de estruturação de dados do TreeModel no CoreQuery 32
Figura 10 – Exemplo do uso do atributo *"Data-"* no CoreQuery 33
Figura 11 – Diagrama de Atividade do fluxo de dados do CoreQuery 34
Figura 12 – Exemplo de declaração de contexto e eventos no HTML através do *Data-Bind*. 35
Figura 13 – Exemplo de declaração de componente no HTML através do *Data-Component*. Note que o *data-bind* é necessário. 36
Figura 14 – Tela de testes. 40
Figura 15 – Código HTML - Angular. 40
Figura 16 – Código Typescript - Angular. 41
Figura 17 – Código TSX - React. 42
Figura 18 – Código HTML - Vuejs. 43
Figura 19 – Código Javascript - Vuejs. 43
Figura 20 – Código HTML - CoreQuery. 44
Figura 21 – Representação gráfica do tamanho (kb) total das aplicações criadas para cada *Framework*. 44
Figura 22 – Quantidade de *requests* feitas durante o carregamento das aplicações. . 45
Figura 23 – Tempo de carregamento (s) das aplicações. 45
Figura 24 – Tempo de *Loading* (ms). 46
Figura 25 – Tempo de *Scripting* (ms). 46
Figura 26 – Tempo de *Rendering* (ms). 47

Lista de quadros

Quadro 1 – Histórico de versões do ECMAScript 18
Quadro 2 – Dados suportados pelo JSON. 27
Quadro 3 – Diferenças entre *framework* e biblioteca 29

Lista de abreviaturas e siglas

AJAX Asynchronous Javascript and XML / Javascript e XML Assíncrono

API Application Programming Interface / Interface de Programação de Aplicação

CSS Cascading Style Sheets / Folha de Estilos em Cascata

DOM Document Object Model / Modelo de Objeto de Documento

ECMA European Computer Manufacturers Association / Associação Européia de Fabricantes de Computadores

HTML HyperText Markup Language / Linguagem de Marcação de Hipertexto

IDE Integrated Development Environment / Ambiente de Desenvolvimento Integrado

JSON JavaScript Object Notation / Notação de Objetos JavaScript

JSX JavaScriptXML ou JavaScript Sintaxe Extension

MVC Model-View-Controller / Modelo-Visão-Controlador

MVVM Model-View-View-Model / Modelo-Visão-Visão-Modelo

MV* Model-View / Familia Modelo-Visão

PHP Hypertext Preprocessor / Pré-processador de Hipertexto

SPA Single Page Application / Aplicações de Página Única

TSX TypescriptXML ou Typescript Sintaxe Extension

UI User Interface / Interface do Usuário

URL Uniform Resource Locator / Localizador Uniforme de Recursos

WHATWG Web Hypertext Application Technology Working Group / Grupo de Trabalho de Tecnologia de Aplicação de Hipertexto da Web

Sumário

1 INTRODUÇÃO 14
1.1 Hipótese 14
1.2 Objetivos 15
1.2.1 Objetivo Geral 15
1.2.2 Objetivos Específicos 15
1.3 Justificativa 15

2 REFERENCIAL TEÓRICO 16
2.1 Frameworks 16
2.1.1 AngularJS 16
2.1.2 ReactJS 16
2.1.3 VueJS 17
2.2 Javascript 17
2.2.1 Evolução do Javascript 17
2.3 Aplicação SPA (*Single Page Application*) 18
2.4 *Templates* 18
2.5 Padrões de Arquitetura MV* 20
2.5.1 MVC: *Model View Controller* 20
2.5.1.1 *Model* 21
2.5.1.2 *View* 21
2.5.1.3 *Controller* 21
2.5.1.4 Vantagens do MVC 22
2.5.2 MVVM: *Model View ViewModel* 22
2.5.2.1 *Model* 23
2.5.2.2 *View* 23
2.5.2.3 *ViewModel* 23
2.5.2.4 Vantagens do MVVM 23
2.6 Diferenças entre MVC e MVVM 24
2.7 HTML 25
2.8 DOM 25
2.9 Typescript 26
2.10 Json - Javascript Object Notation 26
2.11 Expressão Regular 27
2.12 *Observer Pattern* 27
2.13 Programação Reativa 27

2.14 Programação Funcional . . . 28
2.15 Framework x Biblioteca . . . 28
2.16 *Web Components* . . . 29

3 MATERIAIS E MÉTODOS . . . 30
3.1 TreeModel-View: o padrão de arquitetura do CoreQuery . . . 30
3.1.1 TreeModel . . . 30
3.1.2 *View* . . . 31
3.2 Fluxo de dados no Corequery . . . 31
3.3 *View Engine* . . . 31

4 RESULTADOS . . . 32
4.1 Projeto e Implementação . . . 32
4.2 *Template* CoreQuery . . . 34
4.2.1 Eventos . . . 35
4.2.2 Contexto . . . 35
4.2.3 Componentes . . . 35
4.3 API (Application Programming Interface) do Corequery . . . 36
4.3.1 .set() . . . 36
4.3.2 .get() . . . 36
4.3.3 .push() . . . 37
4.3.4 .shift() . . . 37
4.3.5 .pop() . . . 37
4.3.6 .post() . . . 37
4.3.7 .bind() . . . 38
4.3.8 .click() . . . 38
4.3.9 .component() . . . 38
4.4 Testes . . . 38
4.4.1 Elaboração dos Testes . . . 39
4.4.2 Aplicações *Web* Desenvolvidas para os Testes . . . 39
4.4.2.1 Angular . . . 40
4.4.2.2 React . . . 41
4.4.2.3 Vuejs . . . 43
4.4.2.4 CoreQuery . . . 43
4.4.3 Tamanho das Aplicações . . . 44
4.4.4 *Requests* . . . 45
4.4.5 Tempo de Carregamento . . . 45
4.4.6 Loading . . . 46
4.4.7 Scripting . . . 46
4.4.8 Rendering . . . 47

5 CONCLUSÃO 48

REFERÊNCIAS 49

1 Introdução

Em todos os setores da economia existe uma tendência crescente de automação. Diminuir processos e reduzir custos é essencial para a maximização dos lucros e sobrevivência de qualquer empresa na conjuntura econômica atual (CASTRO et al., 2015).

A globalização impulsiona gradativamente a competição empresarial em todos os setores da economia. Na indústria de *software* não é diferente. Dentro do atual contexto mundial é de suma importância que as organizações tenham altos níveis de qualidade e produtividade além de uma preocupação constante na redução dos custos de desenvolvimento (BARROS, 2010).

Uma forma de automatizar e otimizar a produção durante o desenvolvimento de aplicações ou *softwares* é a reutilização de códigos que desempenham funções comuns em várias aplicações distintas. A reutilização de código busca aumentar a produtividade no desenvolvimento evitando a duplicação, aproveitando esforços e experiências do passado e evitando trabalhos repetitivos desnecessários, possibilitando obter produtos com alta qualidade e que sejam economicamente viáveis (SANTOS; CARVALHO, 2015).

Dentre as técnicas de reutilização de códigos destacam-se *frameworks*. Eles são abstrações que unem códigos genéricos e que possuem o objetivo de criar soluções que resolvam problemas comuns entre aplicações distintas durante a etapa de desenvolvimento (ONISHI, 2006). Mediante as diversas soluções existentes atualmente, com o intuito de beneficiar a produção e o desenvolvimento de aplicações *web*, os profissionais buscam constantemente por novas melhorias e meios de otimizar ao máximo a produtividade. Assim, pretende-se com este trabalho obter resposta ao seguinte problema de pesquisa em questão: Como otimizar a produtividade no desenvolvimento da camada de visualização das aplicações *web* em camadas, sem prejudicar a qualidade final do produto?

Neste sentido, surge a proposta de desenvolver um novo *framework*, denominado CoreQuery, destinado a funcionar como uma extensão do HTML que possibilita automatizar a comunicação entre o DOM e a camada de dados. O CoreQuery fornece funções e módulos genéricos para o desenvolvimento de aplicações *web* de forma simplificada. Ele automatiza o fluxo de dados da camada de apresentação e permite que o desenvolvedor ganhe produtividade durante as etapas do desenvolvimento.

1.1 Hipótese

O CoreQuery vai aumentar a produtividade no desenvolvimento e manutenção da camada de visualização de aplicações *web* com reutilização de códigos comuns à várias

aplicações. Além de garantir redução de linhas de códigos em Javascript para definir o comportamento dos elementos HTML se comparado a outros *frameworks* como Angular, React e Vue.

1.2 Objetivos

1.2.1 Objetivo Geral

Desenvolver um *framework* para simplificar o desenvolvimento de aplicações *web* na camada de visualização.

1.2.2 Objetivos Específicos

- Desenvolver funções para renderizar a camada de visualização;
- Desenvolver API de componentes para a interface gráfica;
- Desenvolver funções para detectar alterações no *TreeModel*.

1.3 Justificativa

A capacidade de reuso de código em *frameworks* orientados a objetos permite uma maior produtividade e menor tempo para a comercialização do aplicativo se comparado a forma tradicional de desenvolvimento (MARKIEWICZ; LUCENA, 2001).

O CoreQuery possui como propósito facilitar o desenvolvimento de aplicações *web* fornecendo módulos e funções comuns a manipulação de elementos na camada de visualização, reduzindo assim a quantidade de linhas necessárias a serem programadas se comparada ao modo tradicional de desenvolvimento.

Através de uma única biblioteca Javascript será possível criar componentes para a interface gráfica em HTML5 e automatizar o fluxo de dados na camada de visualização. Assim, será possível que o desenvolvedor foque seus esforços no resultado da aplicação *web* a ser desenvolvida.

- Reutilização de código
  - Aumento da produtividade e,
  - Redução no tempo de comercialização da aplicação.
- Módulos e funções prontas
  - Redução da quantidade de linhas a serem programadas.

2 Referencial Teórico

2.1 Frameworks

Atualmente existem uma vasta variedade de *Frameworks* e bibliotecas disponíveis para o desenvolvimento de aplicações *web single-page*. A seguir uma breve revisão dos mais populares de acordo com a comunidade do Github que serviram de inspiração para o desenvolvimento deste trabalho.

2.1.1 AngularJS

O Angularjs é um *framework* completo desenvolvido em Javascript utilizando o padrão de arquitetura MVC (*model-view-controller*). Com ele é possível criar aplicações *web* em tempo reduzido e de forma facilitada que funcionam bem em qualquer plataforma, *mobile* ou *desktop* (WILLIAMSON, 2015).

O Angular foi criado em 2009 por Miško Hevery e Adam Abrons e é mantido pela equipe do Google, onde Miško Hevery faz parte (BRANAS, 2014). Ele foi utilizado primeiramente em um projeto do Google (o Google Feedback) que em seis meses de desenvolvimento continha 18.000 linhas de código e utilizando o Angular, ele foi replicado e entregue com apenas 1.500 linhas de código, em menos de um mês. Ficou demostrado assim, o grande potencial do Angular que vem ganhando cada vez mais popularidade (SESHADRI; GREEN, 2014).

2.1.2 ReactJS

O React é uma biblioteca para construir UI's que respondem aos eventos de entrada dos usuários, juntamente com a criação e manutenção de estados - usados para preservar as interações dos componentes (SENGUPTA; SINGHAL; CORVALAN, 2016). Ele foi criado por engenheiros do Facebook para solucionar os desafios relacionados ao desenvolvimento de interfaces de usuários complexas, com coleção de dados que eram alterados constantemente (GACKENHEIMER, 2015).

O ReactJS é uma biblioteca versátil, que pode ser combinado com *frameworks* como Angularjs, Ember e Meteor e/ou ainda em junção com outras bibliotecas JavaScript, como KnockoutJS. (ROBBESTAD, 2016).

Com a finalidade de facilitar o uso de sua API de componentes, o React usa uma sintaxe própria, a JSX, que utiliza JavaScript e HTML (HORTON; VICE, 2016). Através de um DOM virtual, o React verifica as partes alteradas, compara com o DOM

real e atualiza apenas o que foi modificado, esta técnica faz com que a aplicação seja carregada mais rapidamente, pois apenas o que foi alterado será reamostrado na página (GACKENHEIMER, 2015).

2.1.3 VueJS

Vue.js é um *framework* desenvolvido por Evan You para construção de Interfaces gráficas de usuário em Javascript. Sua biblioteca principal é focada em atender apenas a camada de visualização. Ele possui uma API simplificada e pode ser integrada com outras bibliotecas ou projetos já existentes (YOU, 2017).

2.2 Javascript

O Javascript foi inventado por Brendan Eich em 1995 e se tornou Padrão ECMA em 1997. ECMA-262 é o nome oficial do Padrão e ECMASscript é o nome oficial da linguagem (W3SCHOOLS, 2017).

O Javascript é uma linguagem orientada a objetos para executar cálculos e manipulações de objetos computacionais dentro do ambiente do hospedeiro (ecma262). Desde seu conceito à concepção, a linguagem Javascript foi desenvolvida para os navegadores de *Internet*. A linguagem tem o intuito de fornecer suporte para interação com o usuário aplicando um estilo de programação orientado a Eventos (KAMBONA; BOIX; MEUTER, 2013).

Em sua versão *Client-Side*, o Javascript permite que conteúdos dinâmicos sejam incluídos em páginas *web*. Assim, *websites* que antes continham apenas conteúdos programados em HTML estático, podem incluir programas que interagem com o usuário, controlam o navegador e criam elementos HTML (FLANAGAN, 2006).

O Javascript é essencial para aplicações *web* modernas. Nos dias atuais, desenvolvedores utilizam crescentemente esta linguagem para escrever aplicações cada vez mais complexas (JENSEN; MOLLER; THIEMANN, 2009).

2.2.1 Evolução do Javascript

Desde de sua padronização em 1997, várias melhorias e mudanças importantes vêm sendo feitas no ECMAScript como, por exemplo, a introdução ao suporte de Expressão Regular em 1999 e suporte a JSON em 2009 (veja tabela a 1) (W3SCHOOLS, 2017).

Quadro 1 – Histórico de versões do ECMAScript

Ano	Nome	Descrição
1997	E.CMAScript 1	Primeira Edição.
1998	ECMAScript 2	Apenas alterações editoriais.
1999	ECMAScript 3	Adicionado suporte a Expressões Regulares. Adicionado {try/catch}.
	ECMAScript 4	Nunca foi lançado.
2009	ECMAScript 5	Adicionado suporte a *"strict mode"*. Adicionado suporte a JSON.
2011	ECMAScript 5.1	Alterações editoriais.
2015	ECMAScript 6 ou ECMAScript 2015	Adicionado suporte a classes e módulos.
2016	ECMAScript 7	Adicionado suporte a operador exponencial (**). Adicionado suporte a *Array.prototype.includes*.

Fonte: W3Schools - 2017

2.3 Aplicação SPA (*Single Page Application*)

Uma aplicação *Single Page* é uma aplicação desenvolvida para navegadores que não recarregam durante o seu uso. Assim, como em todas as aplicações, a aplicação *single page* tem o objetivo de ajudar o usuário a executar uma determinada tarefa (MIKOWSKI; POWELL, 2013).

De acordo com Mikowski (2013),uma SPA consegue fornecer o melhor dos dois mundos - o desempenho de uma aplicação *desktop* e a portabilidade e acessibilidade de um *website*. Aplicações SPA feitas em Javascript são acessíveis de qualquer lugar que exista um navegador disponível, pode ser um computador *desktop*, aparelho *smartphone* ou *tablet*. Elas podem ser abertas em qualquer sistema operacional e não requer nenhum *plugin* proprietário. Possuem ainda a vantagem de poderem ser facilmente atualizadas e distribuídas sem a necessidade de intervenção do usuário.

2.4 *Templates*

Dentro do contexto de desenvolvimento de *frameworks* Javascript, que oferecem o padrão de arquitetura MVC/MV* para o desenvolvimento, é importante colocar em discussão o uso de *Templates* e o relacionamento destes para com a *View* (OSMANI, 2012).

É comum, mas considerado uma má prática, a criação de grandes blocos de marcação HTML através do uso de concatenação de *string*. Desenvolvedores que utilizam desta abordagem, mesclam atributos contendo dados com elementos HTML, sendo obrigados assim, a utilizarem técnicas obsoletas para renderizar a *View*, tal como a função *"document.write"*. Isto significa que a marcação HTML ficará misturada com os códigos escritos

em Javascript, dificultando assim a manutenção, principalmente quando a aplicação em questão possuir complexidade relevante (OSMANI, 2012).

O uso de *template* é geralmente uma solução para o problema apontado. Ele é utilizado para a criação da *View* através de uma linguagem de marcação contendo as variáveis do *template*. Estas podem ser definidas através de uma sintaxe que pode ser interpretada pelo *framework*, que normalmente aceita os dados no formato JSON e realiza o trabalho de criação da *View* automaticamente. Assim, os *templates* podem ser criados e armazenados separadamente e carregados de modo dinâmico quando necessário (OSMANI, 2012). As figuras 1 e 2 são exemplos do uso de *templates* em Aplicações *Web*.

```
1  <li class="pessoa">
2    <h2>{{nome}}</h2>
3    <img src="{{srcfoto}}"/>
4    <div>
5      {{descricao}}
6    </div>
7  </li>
8  
```

Figura 1 – Exemplo de *template* utilizando Handlebars.js.

```
1  <li class="pessoa">
2    <h2><%= nome %></h2>
3    <img src="<%= srcfoto %>"/>
4    <div>
5      <%= descricao %>
6    </div>
7  </li>
8  
```

Figura 2 – Exemplo de *template* utilizando Underscore.js Microtemplates.

É comum confundir *Template* com *View*, porém são abordagens distintas. A *View* é um objeto que observa o *Model* e mantém a representação visual atualizada. O *Template* é uma forma para declarar a especificação de uma *View* para que ela possa ser gerada a partir desta especificação (OSMANI, 2012).

É importante notar para este trabalho, que no desenvolvimento *web* tradicional, a navegação entre *Views* necessita do recarregamento completo da página. Já em uma aplicação Javascript *Single-Page*, os dados são trazidos do servidor através da utilização de

AJAX, sendo possível assim, que a navegação entre as *Views* aconteça sem a necessidade de que a página seja recarregada (OSMANI, 2012).

2.5 Padrões de Arquitetura MV*

Atualmente, dois padrões de arquiteturas são relevantes e bastante utilizados para o desenvolvimento de aplicações *web*: o MVC (*Mode-View-Controller*) e o MVVM (*Model-View-ViewModel*).

No passado, estes padrões de arquiteturas eram comumentes usados na estruturação de aplicativos *desktop* e *server-side*. Nos dias atuais, com a modernização dos navegadores *web*, estes padrões de arquitetura vem cada vez mais sendo utilizado em soluções *front-end* (OSMANI, 2012).

2.5.1 MVC: *Model View Controller*

O MVC é um padrão de arquitetura que propõe uma melhor organização do projeto através da separação de papéis. Ele obriga um isolamento da camada de negócio (*Model*) da camada de visualização (*View*) através de um terceiro componente (*Controllers*) que tradicionalmente é criado para gerenciar as entradas de dados e requisições feitas pelo usuário (OSMANI, 2012).

Este padrão de arquitetura foi criado por Trygve Reenskaug quando trabalhava no Smalltalk-80, uma linguagem de programação da década de 70. Primeiro foi chamado de *Model-View-Controller-Editor* (OSMANI, 2012).

Com o tempo, o padrão de MVC evoluiu e várias novas variações surgiram para atender novas demandas tecnológicas e novas necessidades (SYROMIATNIKOV; WEYNS, 2014).

Atualmente, desenvolvedores Javascript tem uma variedade de *Frameworks* a sua disposição que ajudam a criar aplicações utilizando o padrão de arquitetura MVC (ou variações, aqui referidos como Família MV*) (OSMANI, 2012).

De acordo com Syromiatnikov (2014), a intenção da arquitetura MVC para aplicações *web* é separar a camada de domínio da camada de visualização em três componentes com responsabilidades distintas.

Syromiatnikov (2014) descreve o fluxo de dados no padrão de arquitetura MVC de forma simples: o *Controller* recebe a solicitação do usuário e invoca uma instância do *Model* para posteriormente passar esta para a *View* que por sua vez renderiza a página HTML para que o usuário realize interações e faça novas solicitações.

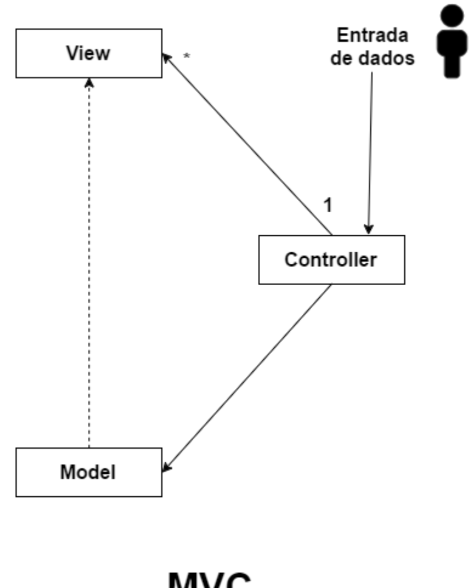

Figura 3 – Padrão de Arquitetura MVC

2.5.1.1 *Model*

O *Model* gerencia os dados de uma aplicação. Ele não possui informações da Interface Gráfica ou da Camada de Apresentação. A sua função é representar o formato em que os dados devem ser apresentados. Quando o *Model* é modificado, ele geralmente notifica seus observadores que uma alteração foi feita para que estes reajam de acordo (OSMANI, 2012).

2.5.1.2 *View*

A *View* é a representação visual do *Model* e representa uma visualização filtrada de seu estado atual. Nas aplicações *web* ela tem como objetivo principal criar e atualizar os elementos HTML. Ela observa o *Model* e é notificada quando este é alterado, permitindo assim atualizar a si mesma de acordo com o necessário (OSMANI, 2012).

Assim, os usuários podem interagir com a *View* sendo possível desta forma ler e editar os valores dos atributos do *Model*. Como a *View* é uma camada de apresentação, os dados geralmente são exibidos em uma interface gráfica "amigável" ao usuário (OSMANI, 2012).

2.5.1.3 *Controller*

O *Controller* nada mais é que a camada de intermediação entre o *Model* e a *View*. Normalmente sua função é atualizar o *Model* quando o usuário interage com a *View* (OSMANI, 2012).

2.5.1.4 Vantagens do MVC

De acordo com Osmani (2014), a separação de conceitos no MVC facilita a modularização das funcionalidades de uma aplicação permitindo:

- Manutenção simplificada. Quando for necessário atualizar a aplicação, tanto o *Model* quanto o *Controller*, será necessário também atualizar a *View* para que corresponda a mudança.
- Desacoplar o *Model* da *View* simplifica testes na lógica de negócio.
- A necessidade de escrever código duplicado do *Model* é eliminado, visto que este pode ser utilizado em várias *Views*.
- A modularização deste tipo de abordagem permite que desenvolvedores responsáveis pela lógica do negócio trabalhem em conjunto com desenvolvedores responsáveis pela interface gráfica.

2.5.2 MVVM: *Model View ViewModel*

O MVVM (*Model View ViewModel*) é um padrão de arquitetura baseado no MVC, que tem a intenção de separar o desenvolvimento de UI (*User-Interfaces*) da camada de negócio e comportamento do projeto. Normalmente, implementações que utilizam esta abordagem utilizam a técnica de declaração de *Data-Bindings* para permitir a separação de trabalho na *View* das outras camadas (OSMANI, 2012).

O padrão de arquitetura MVVM é extremamente desacoplado, visto que a *View* desconhece o *Model*, enquanto o *ViewModel* e o *Model* desconhece a *View*. Da mesma forma o *Model* também desconhece tanto a *View* quanto o *ViewModel* (MOREIRA, 2014).

De acordo com Osmani (2012), isso possibilita o trabalho de desenvolvimento da interface gráfica e do negócio que podem ocorrer quase simultaneamente dentro da mesma base de código por desenvolvedores diferentes. As pessoas responsáveis pela construção do *layout* podem declarar *Bindings* para o *ViewModel* diretamente no documento de marcação (HTML), enquanto o *Model* e o *ViewModel* podem ser mantidos pelos desenvolvedores responsáveis pela lógica da aplicação.

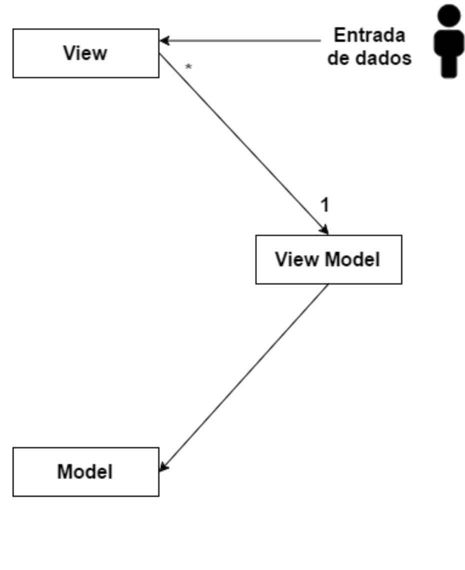

Figura 4 – Padrão de Arquitetura MVVM

2.5.2.1 *Model*

Assim como no padrão de arquitetura MVC, o *Model* no MVVM representa a especificação de domínios dos dados ou informações da aplicação. Eles apenas guardam informações e não lidam com o comportamento da aplicação (OSMANI, 2012).

2.5.2.2 *View*

A *View* possui um comportamento diferente do padrão MVC. No padrão MVVM ela usa o sistema de *bindings* para fazer a comunicação com o *ViewModel* (MOREIRA, 2014).

2.5.2.3 *ViewModel*

O *ViewModel* age como mediador entre a *View* e o *Model*. Ele gerencia o fluxo de dados do *Model* para a *View* e comunica ações realizadas pelo usuário feitas na *View* para o *Model* (MOREIRA, 2014).

2.5.2.4 Vantagens do MVVM

Osmani (OSMANI, 2012) lista algumas vantagens no uso do padrão de arquitetura MVVM para o desenvolvimento de aplicações *web*:

- MVVM facilita significativamente o desenvolvimento em paralelo se a aplicação for feita por mais de um desenvolvedor.

- Abstraindo a *View* deve reduzir significativamente a quantidade de lógica de negócio requerido no *code-behind*.
- O *ViewModel* é mais fácil para ser testado do que programação orientada à eventos.
- O *ViewModel* pode ser executado e testado sem a preocupação com a automação da Interface Gráfica e interação com o usuário.

2.6 Diferenças entre MVC e MVVM

A principal diferença entre estas duas arquiteturas está diretamente ligada a entrada de dados (GEEKSWITHBLOGS, 2009):

- No MVC a entrada de dados é diretamente feita através do *Controller* e não através da *View*. Ela pode ser uma interação com a página ou simplesmente uma requisição feita através de uma URL por exemplo. Em qualquer caso, será o *Controller* que deverá realizar a interligação com alguma funcionalidade.
- No MVVM a entrada de dados é feita diretamente na *View*. O *ViewModel* não possui referência da *View*, assim existe uma maior separação de responsabilidades entre as camadas.

Outra diferença que é importante destacar é o relacionamento entre os componentes (GEEKSWITHBLOGS, 2009):

- No MVC o *Controller* pode selecionar *Views* diferentes para serem renderizadas. A *View* não possui conhecimento ou referência do *Controller*, porém, é necessário que "conheça" o *Model* que lhe será designado pelo *Controller*.
- No MVVM a *View* "conhece" o *ViewModel* mas o *ViewModel* não possui nenhuma referência da *View*. Assim, várias *Views* podem se relacionar com um único *ViewModel*.

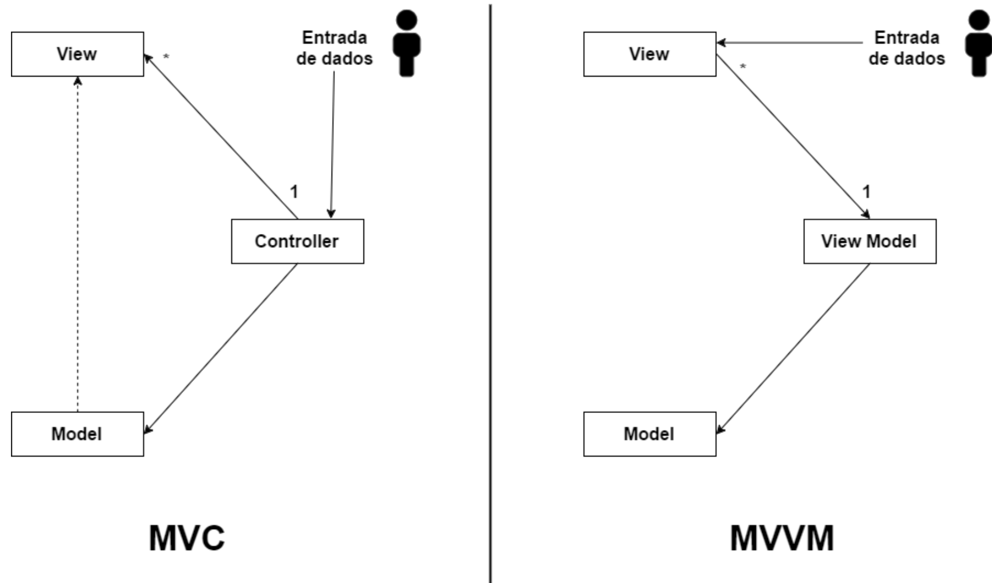

Figura 5 – Diferença entre o padrão de arquitetura MVC e MVVM

2.7 HTML

O HTML ou *HyperText Markup Language* (Linguagem de Marcação de Hipertexto) é uma linguagem de marcação para estruturação e apresentação de conteúdo que define como o navegador irá exibir o conteúdo do documento, como texto e imagens. Ele deixa o documento interativo através de *links* que conectam um documento à outros documentos *web*, em qualquer lugar do mundo (MUSCIANO; KENNEDY et al., 1996).

2.8 DOM

O DOM é uma coleção de objetos que representam os elementos em um documento HTML. Sempre que o navegador renderiza uma página *web*, ele transforma a linguagem de marcação HTML em uma estrutura interna, chamada *Document Object Model* (DOM) (PILGRIM, 2010).

Através de uma API (*Application Programming Interface*) pública é possível inspecionar, criar e modificar dinamicamente os elementos de uma página *web* no navegador utilizando a linguagem Javascript (MEYN et al., 2012).

Um exemplo de hierarquia DOM pode ser visto na imagem abaixo:

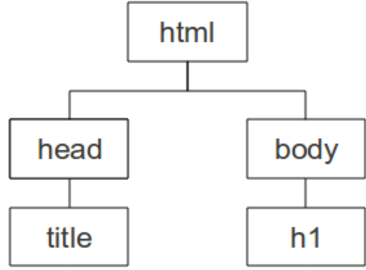

Figura 6 – Exemplo de hierarquia DOM

2.9 Typescript

Typescript é uma extensão do Javascript intencionado em permitir um desenvolvimento de aplicações Javascript de larga-escala de forma mais simples.

Todo código Javascript também é um código TypeScript. O Typescript oferece sistema de módulos, classes, interfaces e um sistema rico de tipagem. A intenção do Typescript é proporcionar uma fácil transição para programadores Javascript. Assim, algoritmos de programação Javascript são suportados sem a necessidade de grandes reescritas de código ou anotações (BIERMAN; ABADI; TORGERSEN, 2014).

Apesar do sucesso do Javascript, ele é uma linguagem deficiente para desenvolver e manter aplicações de grande porte. Typescript é uma extensão criada com intenção de resolver esta deficiência. Ele tenta fornecer uma leve assistência aos programas, os sistemas módulos e de tipos são flexíveis e simples de serem usados. Práticas comuns herdadas do Javascript também podem ser usadas, como por exemplo "*callbacks*". Seu suporte a tipagem ajuda a evitar enganos e permite outros tipos de suporte para o desenvolvimento de aplicativos, como por exemplo sugestões de códigos. O suporte a classes é baseado no padrão do EcmaScript 6 e pode ser exportado para código compatível em EcmaScript 5, caso seja de interesse do desenvolvedor (BIERMAN; ABADI; TORGERSEN, 2014).

O compilador Typescript fornece versões para os principais sistemas operacionais e *plugins* para as principais IDEs de desenvolvimento. Ele possibilita checagem de erros e gera os arquivos Javascript equivalentes para serem executados imediatamente em qualquer ambiente (BIERMAN; ABADI; TORGERSEN, 2014).

2.10 Json - Javascript Object Notation

Javascript Object Notation (JSON), é um formato de texto para a serialização (técnica usada para persistir objetos, ou seja, gravar objetos em disco) de dados estruturados. Ele foi desenvolvido por Douglas Crockford com o objetivo de ser de fácil interpretação, leve e em formato de texto. Foi definido pelo padrão de linguagem de programação ECMAScript

e é derivado de objetos Javascript literais. Seu formato de dados é suportado por todas as linguagens populares de programação como c#, PHP, Java, C++, Python e Ruby (SRIPARASA, 2013).

A estruturação do JSON é simples e tem como objetivo ser mínimo, portátil, textual e ser um subconjunto do Javascript. Ele pode ser representado por quatro tipos primitivos e dois tipos estruturados (CROCKFORD, 2006). Estes estão listados na tabela a seguir.

Quadro 2 – Dados suportados pelo JSON.

Tipos Estruturados	**Tipos Primitivos**
Object	String
Array	Number
	Boolean
	Null

Fonte: Internet Engineering Task Force (IETF) - RFC 4627 - 2006

2.11 Expressão Regular

De acordo com Sidhu (SIDHU; PRASANNA, 2001), Expressão Regular é um padrão que combina com uma ou mais sequência de caracteres. Elas são a chave para processadores de texto flexíveis e eficientes.

Expressões Regulares possuem um padrão de anotação generalizado, que é considerado quase como uma micro linguagem de programação. Ela permite descrever e analisar textos. Com ajuda de ferramentas ou linguagens de programação, ela pode ser usada para adicionar, remover, isolar, separar e cortar qualquer tipo de texto e dados (FRIEDL, 2006).

2.12 *Observer Pattern*

Observer Pattern é uma forma popular de descrever um controle de fluxo baseado em eventos em programas orientados a objeto. Este padrão de arquitetura define a interação entre dois tipos de objetos, observadores e sujeitos, onde o observador reage a qualquer mudança no estado do sujeito. Esta é uma abordagem muito utilizada em *Frameworks* Javascripts para o desenvolvimento aplicações *Single-Page* (NAUMOVICH, 2003).

2.13 Programação Reativa

A Programação Reativa (*Reactive programming*), têm sido declarada como uma alternativa viável ao padrão de arquitetura *Observer* para desenvolvimento de aplicações reativas como interfaces gráficas, animações e sistemas baseados em eventos. A ideia por

trás da programação reativa é suportar abstração em nível de linguagem para sinalizar mudanças nos valores que são atualizados em tempo de execução (SALVANESCHI; MEZINI, 2016).

Na Programação Reativa, desenvolvedores especificam funções dependentes em valores das aplicações para que estes sejam propagados automaticamente quando requerido. Desta forma, não é necessário chamar funções de notificação de atualização dependentes ou observadores (SALVANESCHI; MEZINI, 2016).

2.14 Programação Funcional

Programação funcional é um estilo de programação que trata funções como se fossem dados para facilitar a organização de código e reuso. Em programação funcional, funções podem ser passadas através de argumentos ou retornar uma função como resultado (ELLIOTT, 2014).

2.15 Framework x Biblioteca

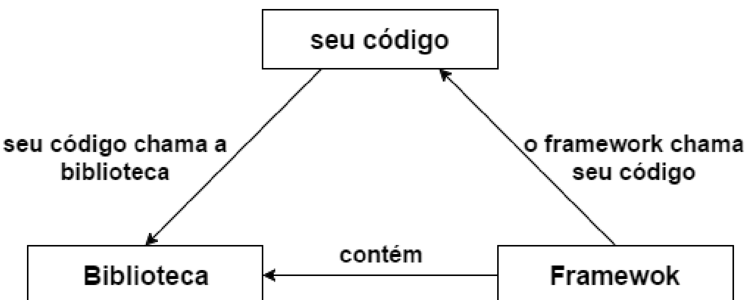

Figura 7 – Representação de interação entre *framework*, biblioteca e código.

Conforme Martin Fowler (FOWLER, 2005), *frameworks* possuem uma característica que os define claramente. Eles ditam as regras em um fenômeno que é chamado de Inversão de Controle.

Inversão de Controle é o elemento chave que faz com que um *framework* seja diferente de uma biblioteca. A biblioteca é basicamente um conjunto de funções ou classes que podem ser chamadas ou usadas. Sempre que essas execuções são finalizadas, o controle é retornado ao seu código (FOWLER, 2005).

Ainda de acordo com Fowler, um *framework* incorpora uma certa abstração de *design*. Ele impõe um modelo a ser seguido. Para utilizá-lo, o desenvolvedor deve adequar-se a este modelo e escrever o código para que ele seja então executado ou chamado pelo *framework*. Veja a tabela 3.

Quadro 3 – Diferenças entre *framework* e biblioteca

Framework	**Biblioteca**
Framework é um design pré definido.	Bibliotecas fazem apenas algumas tarefas específicas.
Tem bibliotecas pré instaladas, sabe qual delas é mais adequada para ele.	Você é quem tem que escolher suas bibliotecas.
O *framework* chama seu código.	Você chama as APIs da biblioteca no seu código.
O controle fica com o *Framework*, caso você esteja utilizando-o.	O controle fica sempre com você.
Fácil de estruturar.	Você precisa estruturar seu projeto manualmente.
O *framework* pode usar várias bibliotecas ou até mesmo linguagens nele.	Bibliotecas talvez tenham dependência em outras bibliotecas, mas não permite múltiplas linguagens.
Desenvolvimento rápido.	Exige um desenvolvimento cauteloso.

Fonte: Inversion Of Control - FOWLER, 2005.

2.16 *Web Components*

O modelo de componentes para *web*, também conhecido como *Web Components*, consiste em quatro partes destinadas que são usadas juntas para permitir que desenvolvedores de aplicações *web* criem *widgets* com maior riqueza visual e que não são possíveis apenas com CSS e de fácil composição e reuso que não são possíveis apenas com Javascript (COONEY; GLAZKOV, 2012).

Estas partes são:

- *Templates*, usados para definir fragmentos de marcação que não tem funcionalidade mas pode ser ativados para o uso posteriormente;
- *Decorators*, no qual é possível aplicação de *templates* para permitir que o CSS enriqueça o visual e comportamento do documento;
- *Custom Elements*, permite desenvolvedores criarem seus próprios elementos, incluindo apresentação e API, que podem ser usados diretamente no documento HTML; e
- *Shadow* DOM, no qual define como a apresentação e comportamento dos *decorators* e *Custom Elements* se encaixam na árvore DOM.

3 Materiais e Métodos

3.1 TreeModel-View: o padrão de arquitetura do CoreQuery

O objetivo geral do CoreQuery é simplificar o desenvolvimento de aplicações *web* na camada de visualização. Para isto, ele será desenvolvido utilizando um padrão de arquitetura em duas camadas criadas com o intuito de reduzir esforços do desenvolvedor para programar a *View*.

Exemplo da arquitetura do padrão TreeModel-View:

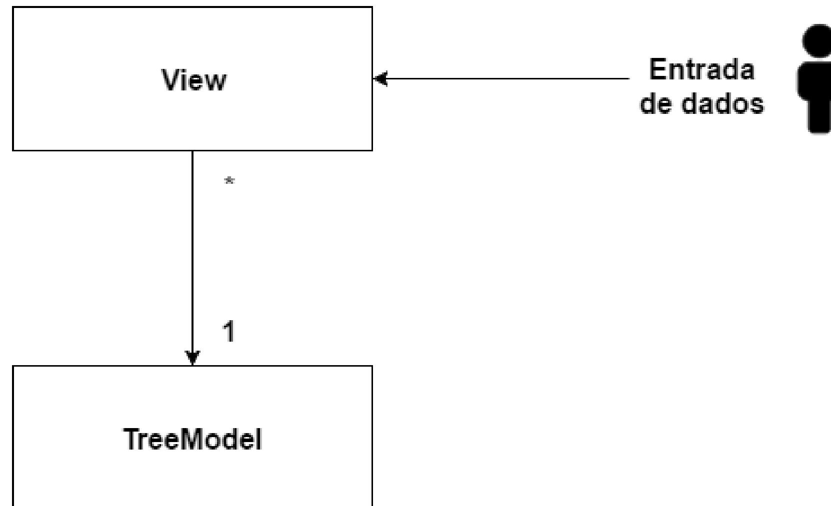

Figura 8 – Padrão de Arquitetura TMV (TreeModel-View)

3.1.1 TreeModel

Assim como no padrão de arquitetura MVC e MVVM, o *TreeModel* representará as especificações de domínio dos dados ou informações da aplicação. Ele também deverá gerenciar o fluxo de dados e informações e não lidará com o comportamento da aplicação. Porém, no padrão de arquitetura *TreeModel-View*, os *Models* devem ser organizados de forma hierárquica, para ser possível que estes sejam acessados por qualquer *View*, em todos os momentos.

No Corequery o *TreeModel* será responsável também pelo gerenciamento do fluxo de dados entre o servidor e o cliente.

3.1.2 *View*

A *View* no padrão de arquitetura *TreeModel-View* tem o intuito de atualizar as informações disponíveis para o usuário no HTML. Ela deverá gerenciar as informações recebidas pelo *TreeModel* e reciclar os elementos correspondentes no DOM. Para isto, será utilizado os atributos do tipo *"data-"* disponíveis nas TAGs HTML. Este tipo de atributo permite armazenar dados no formato de *"string"* diretamente na TAG. No CoreQuery estes atributos serão utilizados para anotar o endereço do contexto do DOM e criar as devidas associações entre a *View* e o *TreeModel*.

3.2 Fluxo de dados no Corequery

O Corequery deverá encapsular o gerenciamento de transmissão de dados na camada *TreeModel*. Para cumprir o proposto, ele utilizará do padrão de arquitetura *Observer* encapsulado em sua estrutura interna para observar cada contexto da árvore hierárquica do *TreeModel*. Esta técnica permite que sempre que algum valor armazenado seja modificado, a *View* seja informada através da função "atualizarView()".

3.3 *View Engine*

Este módulo tem como objetivo o gerenciamento do que será renderizado pelo navegador. Ela deverá conter duas funções principais:

- Gerenciar as entradas de dados e ações feitas pelo usuário e injetar no *TreeModel* definido.
- Receber notificações sobre mudanças do *TreeModel* e criar, atualizar, destruir e adicionar eventos e elementos no DOM do HTML automaticamente, sempre que necessário.

4 Resultados

4.1 Projeto e Implementação

Para o desenvolvimento deste trabalho foi criado o padrão de arquitetura *TreeModel-View*. O termo é referente à suas duas camadas: o *TreeModel* e a *View*. Sua principal característica é a estruturação do *Model* de forma hierárquica, possibilitando assim, utilizar um padrão de estrutura semelhante ao encontrado no DOM do HTML e fácil conversão para o JSON. Veja o exemplo na figura abaixo:

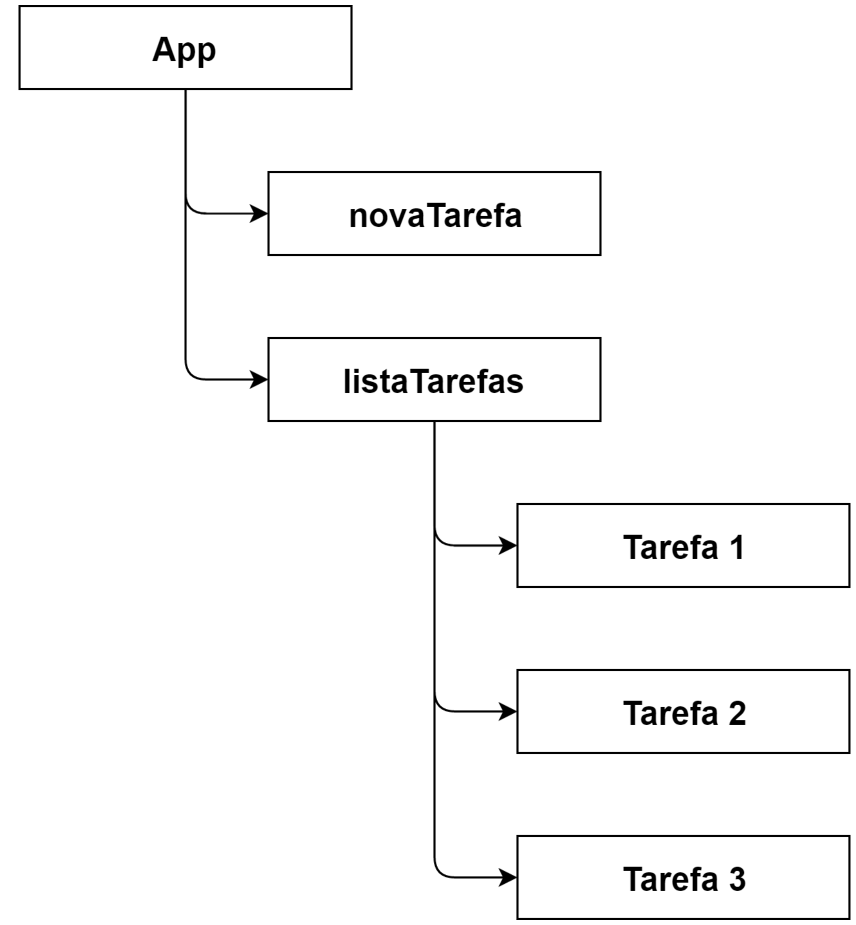

Figura 9 – Exemplo de estruturação de dados do TreeModel no CoreQuery

Este padrão de arquitetura foi desenhado para simplificar o fluxo do *Model* e das *Views* sem a necessidade de um módulo intermediário, como o *Controller* ou a *ViewModel* usado no Angular, no Reactjs e no Vuejs. Assim, em tese, é possível reduzir a complexidade de aplicações que fazem uso deste padrão de arquitetura.

A *View* do CoreQuery interpreta o *Template* e cria as associações entre o DOM e o *TreeModel* definidas através dos atributos do tipo *"Data-"*. Veja o exemplo abaixo:

```
...
<div data-context="pessoa">
    <value type="text" data-value=".nome">
    <value type="text" data-value=".telefone">
</div>
...
```

Figura 10 – Exemplo do uso do atributo *"Data-"* no CoreQuery

No CoreQuery foram definidos dois sentidos de fluxo de dados que ocorrem no *TreeModel*, que são:

- Dados que trafegam no sentido do cliente (navegador) para o servidor;
- Dados que trafegam no sentido do servidor para o cliente (navegador).

No primeiro caso, os dados são criados ou alterados na árvore hierárquica de dados pelos eventos realizados por usuários na *View*, como cliques e digitação, ou solicitações da API Corequery feitas durante a construção pelo desenvolvedor. Esta ação atualiza o *TreeModel* e envia ao servidor dados alterados referentes ao Contexto específico.

No segundo caso, os dados são injetados na árvore hierárquica de dados através de requisições feitas ao servidor. Esta ação atualiza o *TreeModel* e este notifica a *View* sobre a atualização no contexto para que esta responda de acordo.

A figura a seguir apresenta o fluxo de dados do CoreQuery.

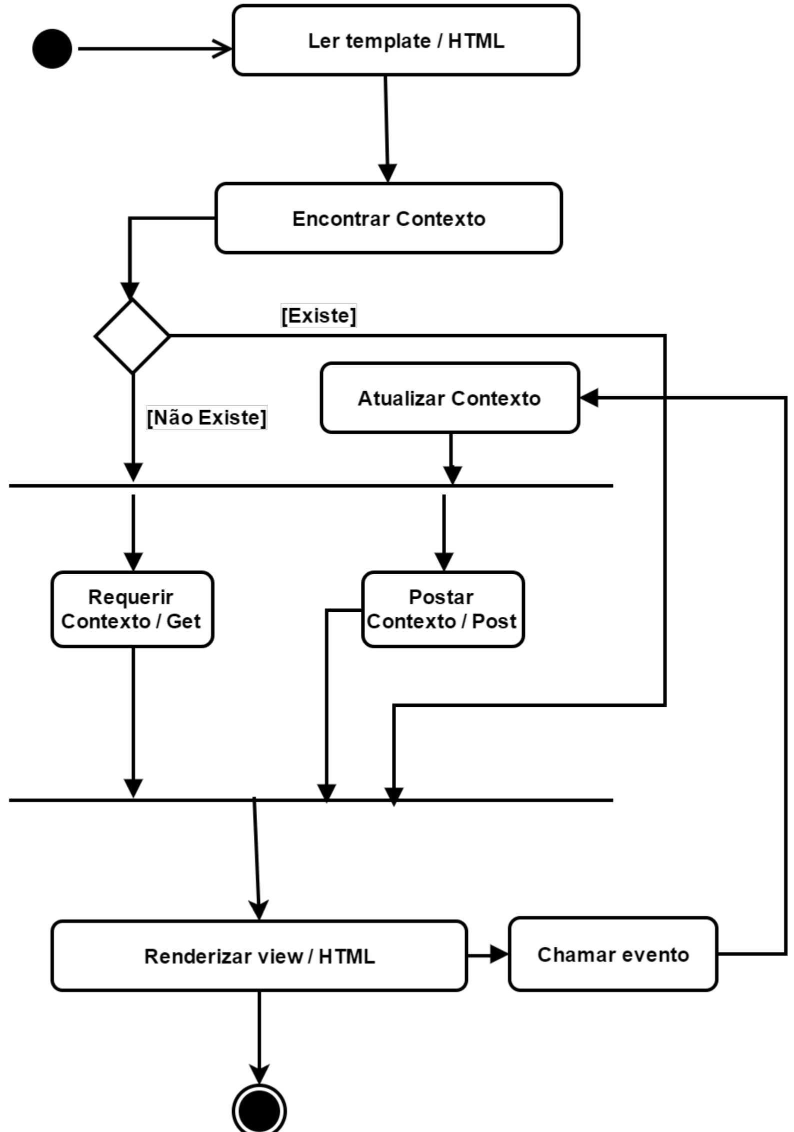

Figura 11 – Diagrama de Atividade do fluxo de dados do CoreQuery

A renderização de *Views* no CoreQuery é um processo simples, que foi planejado para que possa ser feito automaticamente pelo *Framework* sem a necessidade de intervenção do desenvolvedor. Para sua execução não é necessário nem que seja criado um arquivo Javascript por exemplo.

O CoreQuery utiliza programação reativa no *TreeModel*. Sempre que uma alteração for feita na estrutura hierárquica de dados ela irá notificar a camada de visualização para que esta possa atualizar os elementos necessários da aplicação *Single-Page*.

4.2 *Template* CoreQuery

A estruturação da sintaxe de *Template* do Corequery é uma extensão da marcação do HTML convencional. Ele foi planejado para adicionar funcionalidades e definir contextos

nas *Views* através do uso de atributos do tipo “Data”.

Quando estes atributos são definidos, a *View* cria um *Observer* no *TreeModel* para que quando este for modificado ela seja notificada e faça as atualizações necessárias.

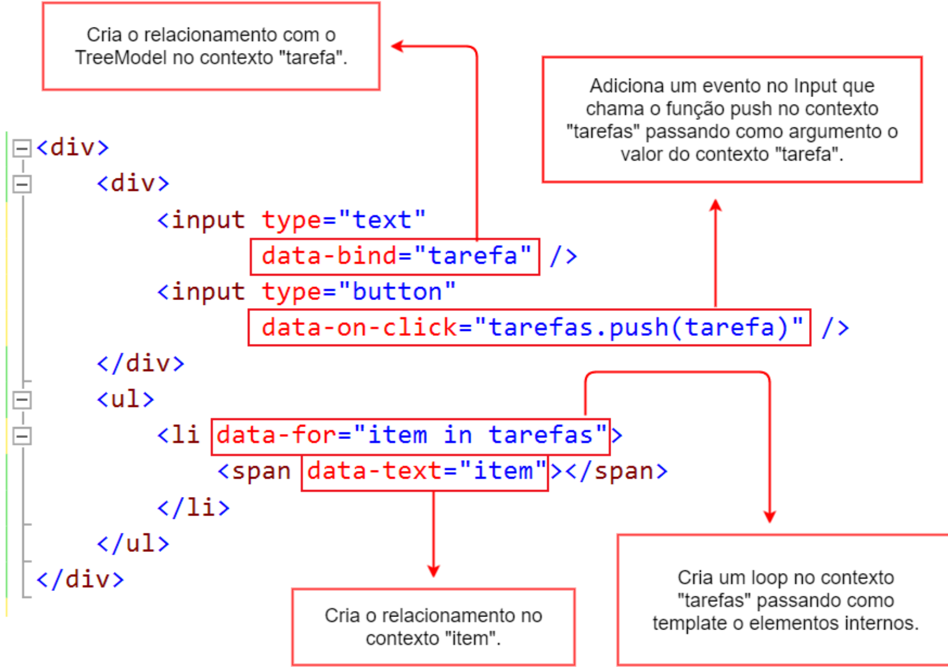

Figura 12 – Exemplo de declaração de contexto e eventos no HTML através do *Data-Bind*.

4.2.1 Eventos

Para definir eventos na *View* através do *Template* é necessário que este seja referenciado através do atributo “Data-” presente no HTML (Ex.: “*data-click*”).

4.2.2 Contexto

O Contexto da *View* possui relação direta com o *TreeModel*. No DOM ele afeta os seus filhos através de herança.

A declaração de um Contexto também é feita através do uso do Atributo “Data” no HTML, dentro da hierarquia do DOM (veja a figura 12).

4.2.3 Componentes

A declaração de componentes no *Template* é feita através do atributo *data-component* na *tag* do elemento desejado no HTML. Veja a figura abaixo:

```
<div>
    <input data-bind="pessoa.nome" />
    <input type="text"
           data-bind="pessoa.nascimento"
           data-component="datepicker" />
</div>
```

Define um elemento como sendo um componente do tipo "datepicker".

Figura 13 – Exemplo de declaração de componente no HTML através do *Data-Component*. Note que o *data-bind* é necessário.

4.3 API (Application Programming Interface) do Corequery

O API do Corequery, inicialmente, irá fornecer ao desenvolvedor as funcionalidades básicas necessárias para gerenciamento da Aplicação *Web* e, se mostrando viável, irá expandir suas funcionalidades.

4.3.1 .set()

Descrição:
Com o método *set* é possível inserir valores no TreeModel em um determinado contexto.

Sintaxe:
Corequery.set(chave: string, valor:any);

Retorno:
null

4.3.2 .get()

Descrição:
Com método *get* é possível requerir valores do TreeModel de um determinado contexto

Sintaxe:
Corequery.get(chave: string);

Retorno:
any

4.3.3 .push()

Descrição:
Com o método *push* é possível inserir valores no TreeModel quando o contexto for *Array*.

Sintaxe:
Corequery.push(chave: string, valor: any);

Retorno:
null

4.3.4 .shift()

Descrição:
Com o método *shift* é possível retirar o primeiro item de um *Array* em um contexto do TreeModel.

Sintaxe:
Corequery.shift(chave: string);

Retorno:
any

4.3.5 .pop()

Descrição:
Com o método *pop* é possível retirar o último item de um *Array* em um contexto do TreeModel.

Sintaxe:
Corequery.pop(chave: string);

Retorno:
any

4.3.6 .post()

Descrição:
Com o método *post* é possível enviar um valor para o servidor no formato JSON.

Sintaxe:
Corequery.push(chave: string, callback: function(valor:any));

Retorno:
null

4.3.7 .bind()

Descrição:
Com o método *bind* é possível anexar um elemento do DOM ao TreeModel.

Sintaxe:
Corequery.bind(elemento: element, chave: string);

Retorno:
null

4.3.8 .click()

Descrição:
Com o método *click* é possível definir um evento no DOM.

Sintaxe:
Corequery.click(elemento: element, callback: function(valor:any));

Retorno:
null

4.3.9 .component()

Descrição:
Com o método *click* é possível criar novos componentes.

Sintaxe:
Corequery.component(nome:string, callback: function(elemento: element, chave:string));

Retorno:
null

4.4 Testes

Alguns testes foram realizados com o objetivo de apresentar os benefícios do Corequery em ralação ao Angularjs, React e Vuejs. A seguir é apresentado desde os ganhos

com redução de linhas de código, quanto tamanho de bibliotecas, quantidade de *Requests*, tempo de carregamento e outros.

4.4.1 Elaboração dos Testes

Os testes foram realizados utilizando o Chrome Devtools do Google. As configurações de *hardware* do computador usado para os teste são:

- Computador: Notebook HP ENVY
- Processador: Inter Core i7-4712HQ CPU @ 2.30GHz
- Memória: 16,0 GB
- Tipo de Sistema: 64-bit
- Sistema operacional: Windows 10

4.4.2 Aplicações *Web* Desenvolvidas para os Testes

Para a elaboração dos testes, foi desenvolvida uma aplicação para cada *Framework* contendo exatamente as mesmas funcionalidades entre elas. Cada uma das aplicações possuem:

- Campo para entrada de valores de texto pelo usuário;
- Botão para adicionar o valor de texto a uma lista;
- Lista com os valores de textos adicionados;
- Botão para exclusão dos valores da lista.

Os testes podem ser reproduzidos através dos repositórios:

- Angularjs: https://github.com/polirs/libraries-tests/tree/master/Angular/app1
- React: https://github.com/polirs/libraries-tests/tree/master/React
- Vuejs: https://github.com/polirs/libraries-tests/tree/master/Vue
- Corequery: https://github.com/polirs/libraries-tests/tree/master/Corequery

Todas as aplicações foram desenvolvidas com base no *layout* abaixo:

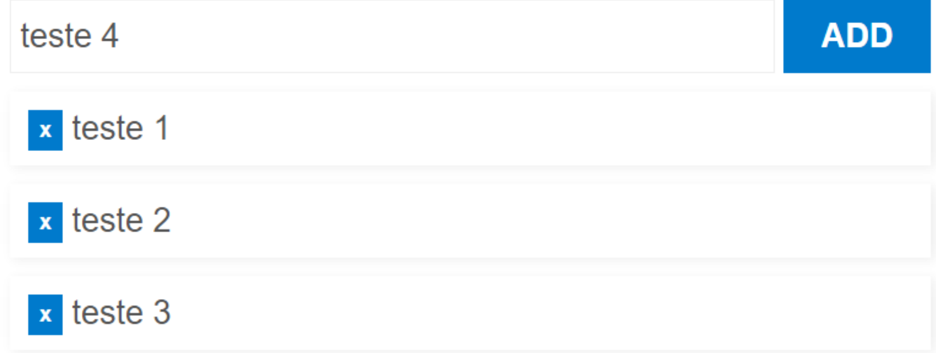

Figura 14 – Tela de testes.

Abaixo podemos conferir e comparar os códigos das aplicações criadas para cada um dos *Frameworks* testados.

4.4.2.1 Angular

Códigos necessários para a criação da aplicação no Angularjs.

```
1  <div id="app">
2    <h1>TODO LIST ANGULAR</h1>
3    <div class="insertItem">
4      <input [ngModel]="item" (keyup)="onKey($event)" type="text" />
5      <button (click)="insertItem()">ADD</button>
6    </div>
7    <div class="itemList">
8      <ul>
9        <li *ngFor="let item of lista; let i = index">
10         <button (click)="deleteItem(i)">x</button>{{ item }}
11       </li>
12     </ul>
13   </div>
14 </div>
```

Figura 15 – Código HTML - Angular.

```
1   import {Component} from '@angular/core';
2   @Component({
3     selector: 'app-root',
4     templateUrl: './app.component.html',
5     styleUrls: ['./app.component.css']
6   })
7   export class AppComponent {
8     item = '';
9     lista = [];
10    disabled = false;
11    insertItem() {
12      this.lista.push(this.item);
13      this.item = "";
14    }
15    deleteItem(item) {
16      this.lista.splice(item, 1);
17    }
18    onKey(event : any) { // without type info
19      this.item = event.target.value;
20    }
21  }
```

Figura 16 – Código Typescript - Angular.

4.4.2.2 React

Para o desenvolvimento da aplicação no React, foi utilizada a biblioteca TSX, que nada mais é que uma biblioteca que reúne Typescript com JSX e que compila JSX (JavaScriptXML) diretamente em JavaScript.

```
class App1 extends React.Component<{}, {item: string, lista: string[]}> {
    constructor(props: {}) {
        super(props);
        this.state = {item:"", lista: []};
    }
    public itemChange = (event) => {
        var newValue = event.target.value;
        this.setState({item:  event.target.value});
    }
    public addClick = (event) => {
        this.state.lista.push(this.state.item);
        this.setState({
            item: "",
            lista: this.state.lista
        });
    }
    public removeItem = (value) => {
        this.state.lista.splice(value, 1);
        this.setState({
            lista: this.state.lista
        });
    }
    render() {
        return (
        <div>
            <h1>TODO LIST REACT</h1>
            <div className="insertItem">
                <input type="text" value={this.state.item} onChange={this.itemChange}  />
                <input value="ADD" type="button" onClick={this.addClick} />
            </div>
            <div className="itemsList">
                <ul>
                    {
                        this.state.lista.map((item, index) => {
                            return <li><input
                            onClick={this.removeItem.bind(this, index)}
                            type="button"
                            value="x" />{item}</li>;
                        })
                    }
                </ul>
            </div>
        </div>);
    }
}
ReactDOM.render(
    <App1  />,
    document.getElementById('app')
);
```

Figura 17 – Código TSX - React.

4.4.2.3 Vuejs

A imagem abaixo mostra o código da aplicação desenvolvida com o Vuejs.

```
1   <div id="app">
2       <h1>TODO LIST VUE</h1>
3       <div class="insertItem">
4           <input v-model="description" type="text" />
5           <input v-on:click="insertItem" type="button" value="ADD" />
6       </div>
7       <div class="itemsList">
8           <ul>
9               <li v-for="(item, index) in items">
10                  <input
11                  v-on:click="removeItem(index)"
12                  type="button"
13                  value="x" />
14                  {{item}}
15              </li>
16          </ul>
17      </div>
18  </div>
```

Figura 18 – Código HTML - Vuejs.

```
1   var app = new Vue({
2       el: '#app',
3       data: {
4           items: []
5       },
6       methods: {
7           insertItem: function(){
8               this.items.push(this.description);
9               this.description = "";
10          },
11          removeItem: function(index, b){
12              this.items.splice(index, 1);
13              console.log(index);
14          }
15      }
16  })
```

Figura 19 – Código Javascript - Vuejs.

4.4.2.4 CoreQuery

Código da aplicação desenvolvida com o CoreQuery.

```
1   <div class="app">
2       <h1>To Do List CoreQuery</h1>
3       <div class="insertItem">
4           <input type="text" data-value="descricao">
5           <input type="button"
6           data-click="lista.push(descricao) descricao.set('')"
7           value="ADD">
8       </div>
9       <div class="itemsList">
10          <ul>
11              <li data-for="lista.i">
12                  <input data-click=".delete()" type="button" value="x" />
13                  <span data-text="."></span>
14              </li>
15          </ul>
16      </div>
17  </div>
```

Figura 20 – Código HTML - CoreQuery.

4.4.3 Tamanho das Aplicações

O primeiro teste realizado mostra a comparação do tamanho total das aplicações após o carregamento inicial de cada uma. Abaixo é possível ver o gráfico que demonstra as diferenças nos tamanhos do CoreQuery e dos *Frameworks* testados.

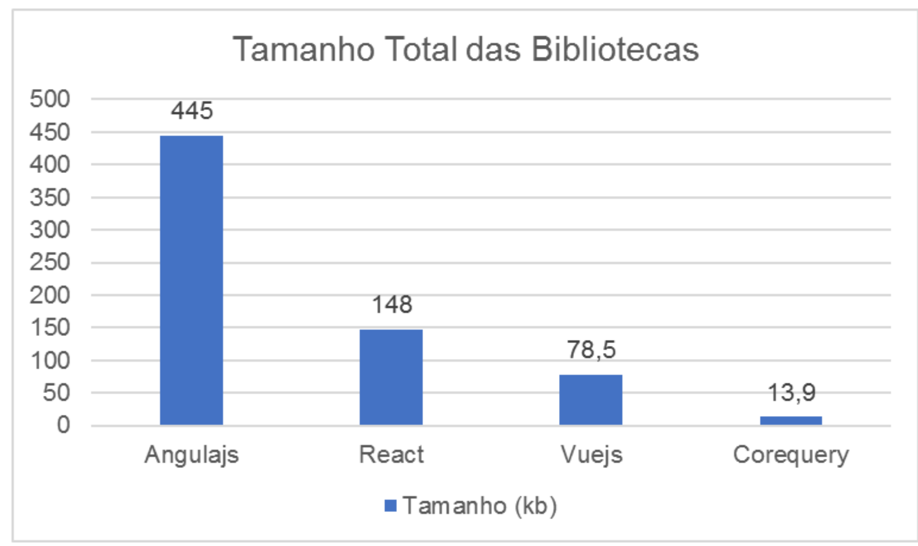

Figura 21 – Representação gráfica do tamanho (kb) total das aplicações criadas para cada *Framework*.

4.4.4 *Requests*

Request é a quantidade de requisições feitas ao servidor pelas aplicações durante o carregamento inicial.

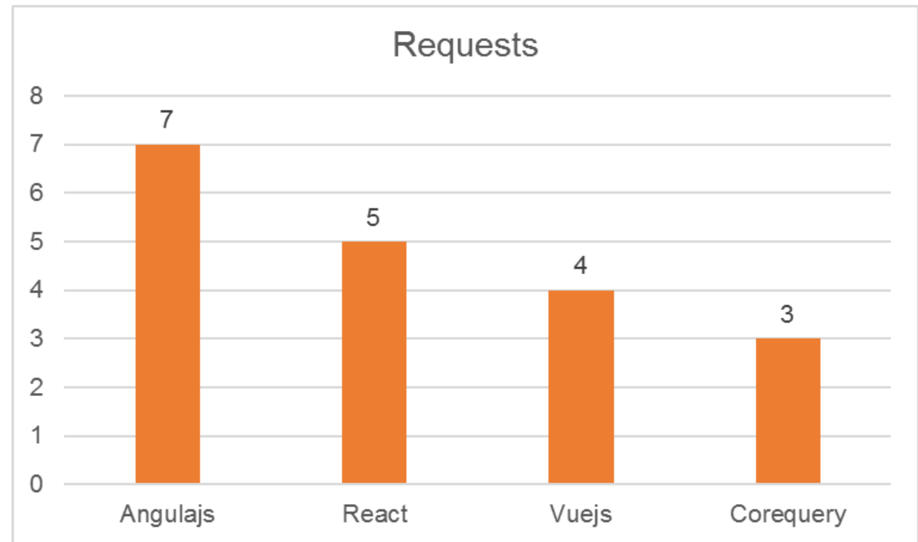

Figura 22 – Quantidade de *requests* feitas durante o carregamento das aplicações.

4.4.5 Tempo de Carregamento

O tempo de carregamento foi medido baseado em uma velocidade de conexão de 4.1Mbps, que é a média de velocidade da internet brasileira atualmente (KNIGHT; FEFERMAN; FODITSCH, 2016).

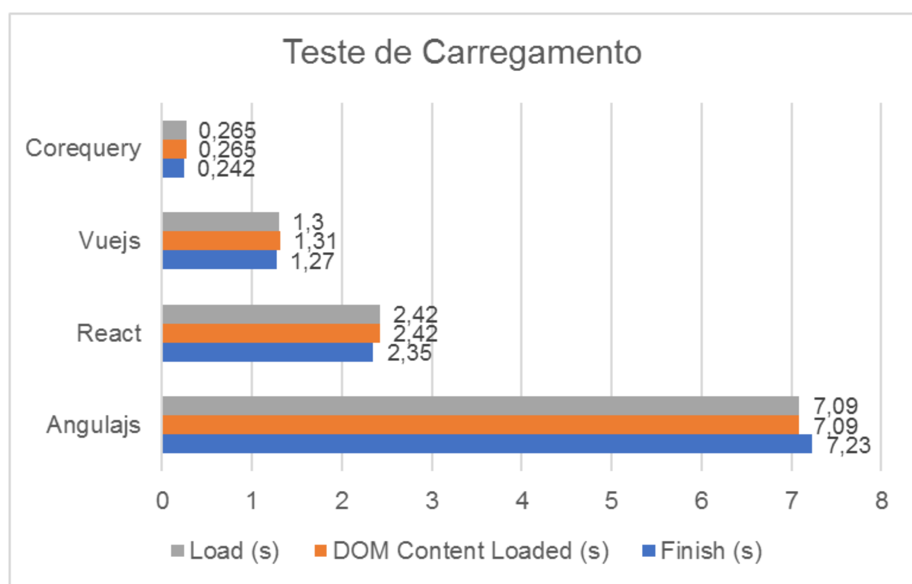

Figura 23 – Tempo de carregamento (s) das aplicações.

4.4.6 Loading

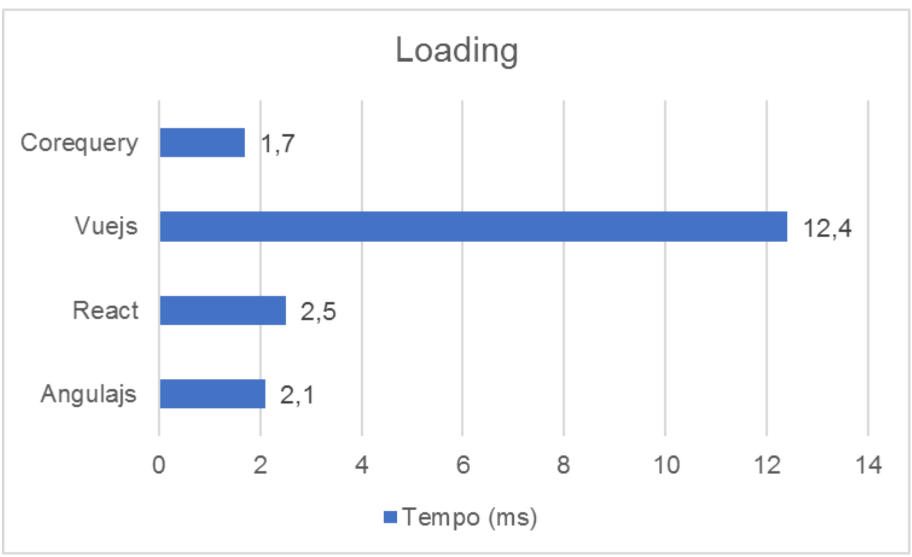

Figura 24 – Tempo de *Loading* (ms).

4.4.7 Scripting

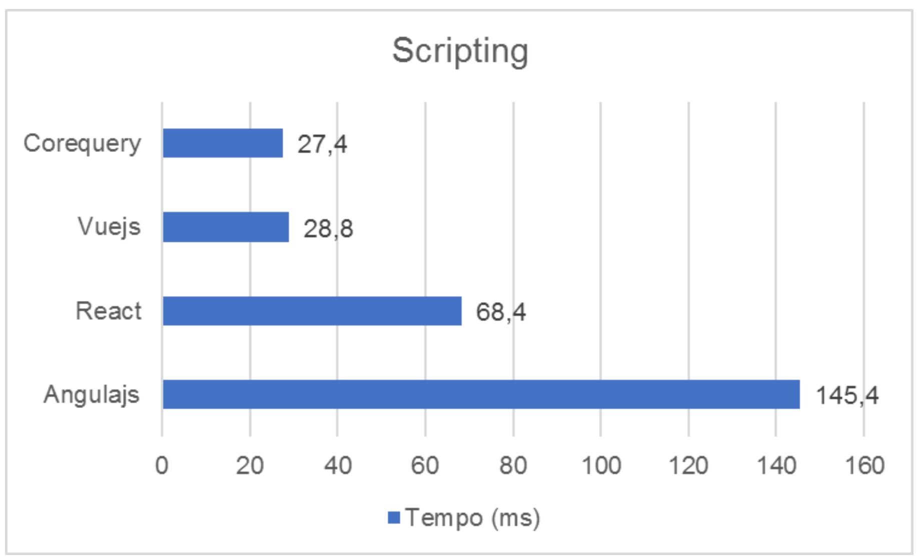

Figura 25 – Tempo de *Scripting* (ms).

4.4.8 Rendering

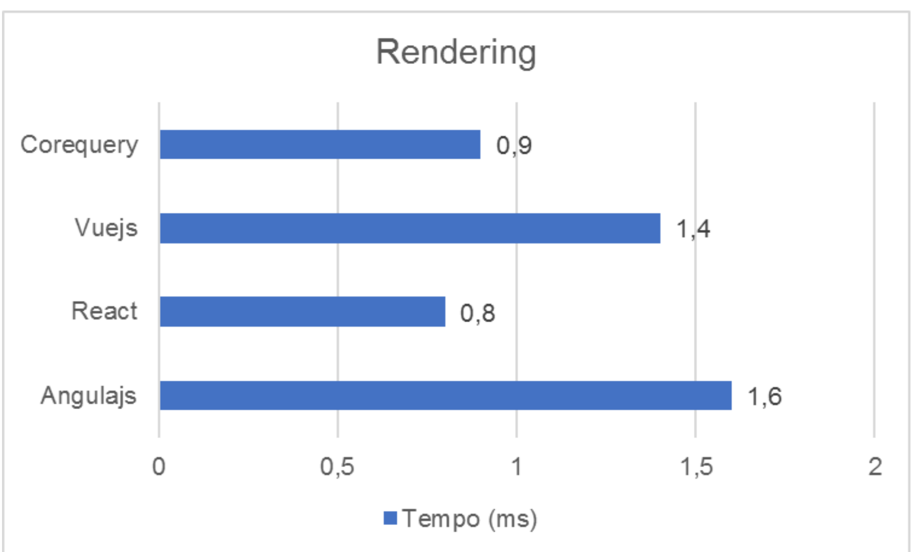

Figura 26 – Tempo de *Rendering* (ms).

5 Conclusão

Mediante os resultados apresentados, conclui-se com esta pesquisa que o CoreQuery oferece benefícios importantes para o desenvolvimento de aplicações *Web Single-Page*. Embora seja um projeto ainda em fase de desenvolvimento e com funcionalidades limitadas comparada a outros *Frameworks*, o CoreQuery mostrou-se eficiente em diversos aspectos, como: facilidade organizacional do formato *TreeModel* na estruturação hierárquica; alternativa eficiente ao *Model* convencional para o gerenciamento dos dados necessários para a renderização da camada de visualização; API eficiente para a criação e manipulação de dados na camada *TreeModel*, na qual, facilita a criação de elementos personalizados na camada de visualização e possibilita que o desenvolvedor integre o CoreQuery a outros *frameworks* e/ou bibliotecas; entre outros.

Nos testes de desempenho foi possível observar que, devido ao seu tamanho, o CoreQuery gastou uma quantidade de tempo menor com o carregamento e execução do *Scripting* inicial da aplicação em relação aos outros *Frameworks* testados. Complementamos também, que a camada de visualização desenvolvida para o CoreQuery possibilita o desenvolvimento de páginas dinâmicas de maneira simples através do uso de atributos próprios no HTML.

Como possíveis trabalhos futuros para enriquecimento deste *Framework* poderão ser implementadas novas funcionalidades, tais como: Método para a definição de Rotas (*Route*); Filtros para listas de dados no *TreeModel*; Operações condicionais no *Template*; Bibliotecas para renderização *Server-side*.

Referências

BARROS, E. A. d. Catálogo de fatores que influenciam a produtividade no desenvolvimento de software. Universidade Federal de Pernambuco, 2010. Citado na página 14.

BIERMAN, G.; ABADI, M.; TORGERSEN, M. Understanding typescript. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 2014. p. 257–281. Citado na página 26.

BRANAS, R. *AngularJS Essentials*. [S.l.]: Packt Publishing Ltd, 2014. Citado na página 16.

CASTRO, C. et al. A gestão estratégica de custos como diferencial competitivo para micro e pequenas empresas. *Gestão em foco-UNISEPE*, v. 1, p. 1–10, 2015. Citado na página 14.

COONEY, D.; GLAZKOV, D. Introduction to web components. *Working draft, W3C*, 2012. Citado na página 29.

CROCKFORD, D. The application/json media type for javascript object notation (json). 2006. Citado na página 27.

ELLIOTT, E. *Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries*. [S.l.]: "O'Reilly Media, Inc.", 2014. Citado na página 28.

FLANAGAN, D. *JavaScript: the definitive guide*. [S.l.]: "O'Reilly Media, Inc.", 2006. Citado na página 17.

FOWLER, M. *InversionOfControl*. 2005. Disponível em: <https://martinfowler.com/bliki/InversionOfControl.html>. Citado na página 28.

FRIEDL, J. E. *Mastering regular expressions*. [S.l.]: "O'Reilly Media, Inc.", 2006. Citado na página 27.

GACKENHEIMER, C. *Introduction to React*. [S.l.]: Apress, 2015. Citado 2 vezes nas páginas 16 e 17.

GEEKSWITHBLOGS. *MVVM Compared To MVC and MVP*. 2009. Disponível em: <http://geekswithblogs.net/dlussier/archive/2009/11/21/136454.aspx>. Citado na página 24.

HORTON, A.; VICE, R. *Mastering React*. [S.l.]: Packt Publishing Ltd, 2016. Citado na página 16.

JENSEN, S. H.; MOLLER, A.; THIEMANN, P. Type analysis for javascript. In: SPRINGER. *International Static Analysis Symposium*. [S.l.], 2009. p. 238–255. Citado na página 17.

KAMBONA, K.; BOIX, E. G.; MEUTER, W. D. An evaluation of reactive programming and promises for structuring collaborative web applications. In: ACM. *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. [S.l.], 2013. p. 3. Citado na página 17.

KNIGHT, P.; FEFERMAN, F.; FODITSCH, N. *Broadband In Brazil: past, present, and future*. [S.l.]: Google Access, 2016. Citado na página 45.

MARKIEWICZ, M. E.; LUCENA, C. J. de. Object oriented framework development. *Crossroads*, ACM, v. 7, n. 4, p. 3–9, 2001. Citado na página 15.

MEYN, A. J. et al. Browser to browser media streaming with html5. 2012. Citado na página 25.

MIKOWSKI, M. S.; POWELL, J. C. Single page web applications. *B and W*, 2013. Citado na página 18.

MOREIRA, R. M. L. de M. Pattern-based gui testing. Citeseer, 2014. Citado 2 vezes nas páginas 22 e 23.

MUSCIANO, C.; KENNEDY, B. et al. *HTML, the definitive Guide*. [S.l.]: O'Reilly & Associates, 1996. Citado na página 25.

NAUMOVICH, G. Using the observer design pattern for implementation of data flow analyses. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 28, n. 1, p. 61–68, 2003. Citado na página 27.

ONISHI, A. *Técnicas de Reuso de Software aplicados na elaboração de Arquiteturas Corporativas*. [S.l.]: Universidade de São Paulo,[200-], 2006. Citado na página 14.

OSMANI, A. *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. [S.l.]: "O'Reilly Media, Inc.", 2012. Citado 6 vezes nas páginas 18, 19, 20, 21, 22 e 23.

PILGRIM, M. *HTML5: Up and Running: Dive into the Future of Web Development*. [S.l.]: "O'Reilly Media, Inc.", 2010. Citado na página 25.

ROBBESTAD, S. A. *ReactJS Blueprints*. [S.l.]: Packt Publishing Ltd, 2016. Citado na página 16.

SALVANESCHI, G.; MEZINI, M. Debugging for reactive programming. In: ACM. *Proceedings of the 38th International Conference on Software Engineering*. [S.l.], 2016. p. 796–807. Citado na página 28.

SANTOS, A. H. dos; CARVALHO, N. R. Frameworks e seus benefícios no desenvolvimento de software. 2015. Citado na página 14.

SENGUPTA, D.; SINGHAL, M.; CORVALAN, D. *Getting Started with React*. [S.l.]: Packt Publishing Ltd, 2016. Citado na página 16.

SESHADRI, S.; GREEN, B. *AngularJS: Up and Running: Enhanced Productivity with Structured Web Apps*. [S.l.]: "O'Reilly Media, Inc.", 2014. Citado na página 16.

SIDHU, R.; PRASANNA, V. K. Fast regular expression matching using fpgas. In: IEEE. *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. [S.l.], 2001. p. 227–238. Citado na página 27.

SRIPARASA, S. S. *JavaScript and JSON Essentials*. [S.l.]: Packt Publishing Ltd, 2013. Citado na página 27.

SYROMIATNIKOV, A.; WEYNS, D. A journey through the land of model-view-design patterns. In: IEEE. *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*. [S.l.], 2014. p. 21–30. Citado na página 20.

W3SCHOOLS. *Versões Javascript*. 2017. Disponível em: <https://www.w3schools.com/js/js_versions.asp>. Citado na página 17.

WILLIAMSON, K. *Learning AngularJS: A Guide to AngularJS Development*. [S.l.]: "O'Reilly Media, Inc.", 2015. Citado na página 16.

YOU, E. *Vue.js - The Progressive JavaScript Framework*. 2017. Disponível em: <https://vuejs.org/>. Citado na página 17.