



YAN MAGALHÃES LEITE

FLUX:

Arquitetura de Aplicações *Client-side*

Belo Horizonte
2016

YAN MAGALHÃES LEITE

FLUX:

Arquitetura de Aplicações *Client-side*

Trabalho de Conclusão de Curso
apresentado à Faculdade Pitágoras, como
requisito parcial para a obtenção do título
de graduado em Ciências da
Computação.

Orientador:

Belo Horizonte

2016

FLUX:

ARQUITETURA DE APLICAÇÕES *CLIENT-SIDE*

Trabalho de Conclusão de Curso apresentado à Faculdade Pitágoras, como requisito parcial para a obtenção do título de graduado em Ciências da Computação.

Aprovado em: 06/12/2016

BANCA EXAMINADORA

Prof^(a). Cristiano Nunes

Prof^(a). Gilberto Giacomini

Prof^(a). Titulação Nome do Professor(a)

Dedico este trabalho a meus amigos,
família e colegas de trabalho da área
de desenvolvimento web.

AGRADECIMENTOS

Agradeço a todos os meus colegas de classe por todo o apoio durante a confecção deste trabalho. Aos colegas de trabalho e comunidade de desenvolvedores, que me possibilitaram esclarecer as minhas dúvidas durante a confecção deste trabalho. Aos meus amigos e familiares por toda a cooperação e apoio.

RESUMO

A arquitetura de aplicações é elemento fundamental para se criar softwares de qualidade e que permitam manutenção facilitada e maior agilidade na implementação de novas funcionalidades. É mencionado a evolução das tecnologias utilizadas para o desenvolvimento de aplicações para a internet, e que a medida que novas e mais complexas aplicações surgem, faz-se necessário a utilização de uma arquitetura e de se seguir os princípios de engenharia de software. Por muitos anos, o modelo MVC foi o padrão mais adotado na escolha para a implantação de aplicações web e softwares em geral. Contudo, em 2014, o Facebook apresentou o Flux, um novo modelo de arquitetura para o desenvolvimento de aplicações client-side, que desde então, tem sido utilizado por muitas aplicações e a sua adoção por parte da comunidade de desenvolvedores foi imediata e bastante atrativa. Sua arquitetura permite um único fluxo direcional de dados, diferente do MVC, que em suas implementações nas aplicações client-side, possibilita um fluxo bidirecional e alterações em cascata em toda a aplicação. É apresentado a fundo todo o funcionamento da arquitetura Flux e os benefícios que ela traz para o desenvolvimento de aplicações client-side, ressaltando-se a importância e relevância do padrão MVC, uma vez que os benefícios da utilização da arquitetura Flux ocorrem em cenários específicos.

Palavras-chave: Flux; MVC; Arquitetura; Front-End; Fluxo Único de Dados.

ABSTRACT

The architecture of applications is fundamental element to create quality software and that allows easy maintenance and greater agility in the implementation of new functionalities. It is mentioned the evolution of the technologies used for the development of applications for the Internet, and that as new and more complex applications arise, it becomes necessary to use an architecture and to follow the principles of software engineering. For many years, the MVC model was the most widely adopted standard in the deployment of web applications and software in general. However, in 2014 Facebook introduced Flux, a new architecture model for the development of client-side applications, which has since been used by many applications and its adoption by the developer community was immediate and quite attractive . Its architecture allows a single directional flow of data, different from MVC, which in its implementations in the client-side applications, allows a bidirectional flow and cascade changes throughout the application. The full functionality of the Flux architecture is presented and the benefits it brings to the development of client-side applications, highlighting the importance and relevance of the MVC standard, since the benefits of using the Flux architecture occur in specific scenarios.

Key-words: Flux; MVC; Architecture; Front-End; Single Unidirectional Data Flow

LISTA DE ABREVIATURAS E SIGLAS

| | |
|-------|---------------------------------------|
| MVC | <i>Model-View-Controller</i> |
| JS | Javascript |
| SPA | Single Page Application |
| F8 | Conferência de Tecnologia do Facebook |
| DOM | Document Object <i>Model</i> |
| W3C | World Wide Web Consortium |
| HTML | HyperText Markup Language |
| HTTPS | HyperText Transfer Protocol Secure |

LISTA DE FIGURAS

| | |
|--|----|
| Figura 1 – Processo de pedido e resposta de uma página web..... | 14 |
| Figura2 - A evolução do software | 17 |
| Figura 3: Diagrama do modelo MVC..... | 21 |
| Figura4: Diagrama de funcionamento do modelo MVP..... | 23 |
| Figura5– Diagrama de funcionamento do Modelo MVVM. | 24 |
| Figura 6: Complexidade de uma aplicação feita no padrão MVC..... | 27 |
| Figura7 – Uma nova visão sobre a complexidade de uma aplicação a medida que ela cresce..... | 27 |
| Figura 8: Diagrama do funcionamento da estrutura Flux..... | 29 |
| Figura 9: Representação arquitetura flux e o fluxo único de dados..... | 30 |
| Figura 10: Descritivo de uma nova ação realizada na View | 30 |
| Figura11 – Aplicações flux complexas..... | 31 |
| Figura 12 – Representação do projeto Flux hospedado no Github..... | 40 |
| Figura 13 – Representação do Projeto Flux Comparasion..... | 41 |
| Figura 14 – Lista de bibliotecas que implementam a arquitetura Flux..... | 41 |
| Figura 15 – Representação do projeto Redux no Github..... | 43 |
| Figura 16: Registro do Dispatcher, (Elaborado pelo autor) | 47 |
| Figura 17: Mapeamento das ações da aplicação, (Elaborado pelo autor)..... | 47 |
| Figura 18: Exemplificação do dispatch das actions, (Elaborado pelo autor)... | 48 |
| Figura 19: Criação da Store..... | 49 |
| Figura 20: Realização das ações e emissão de eventos após atualização de valores na Store..... | 50 |
| Figura 21: Realização das ações e emissão de eventos após atualização de valores na Store..... | 51 |
| Figura 22: Realização das ações e emissão de eventos após atualização de valores na Store..... | 52 |

LISTA DE TABELAS

| | |
|--|----|
| Quadro 1 – Comparativo entre as arquiteturas MVC e Flux..... | 45 |
|--|----|

SUMÁRIO

| | | |
|-----------|---|-----------|
| 1 | INTRODUÇÃO | 12 |
| 1.1 | Problema de Pesquisa..... | 15 |
| 1.2 | Objetivos do Trabalho..... | 15 |
| 1.2.1 | Objetivo Geral | 15 |
| 1.2.2 | Objetivos Específicos | 15 |
| 2. | FUNDAMENTAÇÃO TEÓRICA | 17 |
| 2.1 | O FUNCIONAMENTO DA WEB | 17 |
| 2.1.1 | História | 17 |
| 2.1.2 | Estrutura e Funcionamento | 18 |
| 2.1.3 | Tecnologias utilizadas e evolução das aplicações <i>Client-side</i> | 20 |
| 2.2. | ARQUITETURA DE PROJETOS DE SOFTWARE..... | 21 |
| 2.2.1 | O padrão de arquitetura MVC | 23 |
| 2.2.1.1 | Funcionamento..... | 24 |
| 2.2.1.2 | Características | 26 |
| 2.2.1.3 | Variações do MVC | 27 |
| 2.2.1.3.1 | MVP | 28 |
| 2.2.1.3.2 | MVVM | 29 |
| 2.2.1.4 | MVC em aplicações <i>client-side</i> | 30 |
| 2.3 | O surgimento do Flux..... | 31 |
| 2.4 | A ARQUITETURA FLUX..... | 35 |
| 2.4.1 | Fluxo único de dados | 35 |
| 2.4.2 | Dispatcher | 37 |
| 2.4.3 | Actions | 38 |
| 2.4.4 | Store..... | 38 |
| 2.4.5 | Views..... | 39 |
| 2.4.5.1 | A biblioteca React | 39 |
| 2.5 | CARACTERÍSTICAS | 40 |
| 2.6 | ADOÇÃO DO FLUX..... | 41 |
| 2.7 | MVC X FLUX | 45 |
| 3. | METODOLOGIA | 48 |
| 4. | ESTUDO DE CASO | 49 |
| 5. | DISCUSSÃO E RESULTADOS | 55 |
| 6. | CONSIDERAÇÕES FINAIS | 56 |

| | |
|-------------------------|-----------|
| REFERÊNCIAS..... | 57 |
|-------------------------|-----------|

1 INTRODUÇÃO

Como se sabe, a criação de um conjunto de instruções que serão interpretadas por um computador, que realizam tarefas específicas, sempre foi e sempre será complexo. Este conjunto de introduções, também conhecidos como softwares ou programas de computadores, são capazes de resolver problemas específicos, e desenvolver softwares de qualidade, quer seja um novo produto, ou apenas uma pequena parte de um programa de computador já existente, e que se gere o valor esperado para os seus utilizadores, depende de inúmeros fatores, circunstâncias e decisões, que vão muito além das tecnologias utilizadas.

O autor afirma que definir e planejar toda a estrutura de armazenamento dos dados a serem utilizados pelo programa (conhecido como banco de dados), desenvolver toda a base para visualização dos dados e ter total entendimento problema, são alguns elementos que estão em torno da criação de um software. Bragger (2013) relata que além disso, escrever um conjunto de códigos que antecipe uma evolução do programa, de modo que ele possa se comportar de forma escalável, ou seja, sem demandar a compra de mais servidores (computadores mais robustos com alta capacidade de processamento) a cada aumento de acessos simultâneos ao sistema, faz com que o ecossistema em torno da criação de softwares seja enorme e bastante complexo.

Ao se tratar de softwares que deveram ser executados na internet, é impossível não notar a quantidade de programas utilizados diariamente, como também a qualidade dos programas desenvolvidos. Pode-se citar as redes sociais como exemplo, grandes softwares que são executados na internet. Isso só foi possível graças a evolução de todas as tecnologias necessárias para a criação de softwares para a internet.

O autor também afirma que o momento atual é de grande evolução das ferramentas e tecnologias necessárias para o desenvolvimento voltado a internet. Toda essa evolução trouxe mais poder para as aplicações web, principalmente do lado do navegador (conhecido como browser). Muitas

aplicações se tornaram possíveis apenas com estas tecnologias, como também se percebe um aumento significativo dos sistemas voltados para a internet. Sendo assim, é essencial para se desenvolver uma aplicação web de qualidade, pensar em todo o ecossistema de desenvolvimento de software e de tecnologias voltadas ao front-end (termo que descreve as interfaces visuais, exibidas pelo browser), como nas tecnologias de *back-end* (termo que descreve toda a parte interna do funcionamento de uma página web) do lado do servidor.

De acordo com Fowler (2002), para se criar softwares, recomenda-se a utilização de alguns padrões, para tornar o software mais robusto, passível de evolução e que não se exija tanto esforço dos profissionais para a realização de manutenções. Se faz necessário então, adotar práticas e padrões relacionados a arquitetura do software. Por muitos anos, e ainda hoje, o *modelo* de arquitetura em camadas foi um dos mais utilizados no desenvolvimento de sistemas. Praticamente todas as linguagens de programação de computadores possuem ferramentas, frameworks e bibliotecas (estes são conjuntos de códigos que facilitam a criação de softwares) baseados no *modelo* MVC (*Model-View-Controller*), que é um destes tipos arquitetura em camadas. Este consagrado *modelo*, criado em 1970, se estendeu não somente para as aplicações *back-end*, mas também para o desenvolvimento de aplicações front-end.

Em 2014, o Facebook¹, uma das maiores empresas de tecnologia da atualidade, apresentou na F8 *Developer Conference* (sua conferência onde seu time de engenharia apresenta as novidades da empresa) que eles estavam sofrendo alguns problemas com a arquitetura de seus sistemas, do lado do front-end, e que pelo próprio funcionamento da arquitetura MVC, eles não estavam conseguindo implementar novas funcionalidades de uma maneira mais rápida, e com qualidade.

Neste evento, eles apresentaram um novo *modelo* de arquitetura, que havia substituído o MVC e que estava sendo utilizado em suas soluções, chamado de Flux. Com este *modelo*, a arquitetura permitia um único fluxo

¹ Disponível em <https://facebook.github.io/flux/docs/overview.html>.

direcional de dados, diferente do MVC, que por sua vez possibilitava um caminho bidirecional para o fluxo das informações. O Flux ganhou rápida adesão por parte da comunidade de desenvolvedores e profissionais que atuam no desenvolvimento de aplicações web, pelo fato de uma empresa como o Facebook² estar utilizando um novo *modelo* recém-criado em seus produtos, e não terem optado por um padrão consolidado como o MVC.

Esta popularidade também foi fortemente influenciada pela adesão e evolução do React, uma biblioteca para a criação de componentes (partes de uma interface web) criada pelo próprio Facebook, que é utilizada em seu site e também no Instagram, uma outra rede social, também mantida pela empresa. Apesar do pouco tempo de criação, o Flux já inspirou a criação de algumas bibliotecas que implementam a sua arquitetura, e a cada dia o número de aplicações que usam essa arquitetura aumentam.

Neste trabalho, vamos conhecer a fundo como funciona a arquitetura Flux, quais os benefícios ela pode trazer em uma aplicação *client-side*, descobrir as razões que fizeram o time de Engenharia do Facebook utilizá-lo no lugar do MVC e identificar por quê esta arquitetura tem sido tão utilizada e discutida pelos profissionais de desenvolvimento de aplicações web.

O motivo da realização deste trabalho se deve ao fato do reconhecimento da importância da arquitetura no desenvolvimento de software e descobrir porque o Flux se tornou uma alternativa tão interessante para o desenvolvimento de aplicações web.

Para a realização deste trabalho, será feito um estudo profundo sobre o funcionamento desta nova arquitetura, apresentando comparativos entre ela e o modelo MVC. Também será apresentado um estudo de caso feito pela equipe do Facebook sobre como eles utilizaram o Flux em sua plataforma de chat, e também será desenvolvido um sistema, utilizando a arquitetura Flux, que será feito pelo próprio autor.

² Disponível em <https://facebook.github.io/flux/docs/overview.html>

1.1 PROBLEMA DE PESQUISA

Quais os desafios de implantar e manter uma aplicação web, que utiliza um *modelo* de arquitetura em camadas? Quais foram as evoluções em termos de arquiteturas adotadas para aplicações web? Por quê utilizar o Flux ao invés do MVC, ou qualquer outro padrão já existente, para o desenvolvimento de aplicações *client-side*?

1.2 OBJETIVOS DO TRABALHO

O desenvolvimento deste trabalho visa a obtenção dos seguintes objetivos:

1.2.1 Objetivo Geral

Apresentar a fundo o funcionamento da arquitetura Flux, criada e utilizada hoje pelo Facebook e descobrir seus benefícios no desenvolvimento de aplicações *client-side*.

1.2.2 Objetivos Específicos

Destacar e enfatizar a importância da arquitetura do desenvolvimento de software;

Descrever os desafios encontrados em uma aplicação web que utiliza o *modelo* de arquitetura em camadas;

Apresentar o padrão de arquitetura MVC e suas variações;

Entender os problemas que o Facebook possuía com a arquitetura em camadas;

Apresentar a fundo o funcionamento da arquitetura Flux;

Informar sobre o *Redux*, uma biblioteca que possibilita uma maneira mais fácil de implementar a arquitetura Flux no desenvolvimento de aplicações web;

Apresentar a biblioteca *React*, criada e mantida pelo time de engenharia do Facebook, que permite a criação de componentes para uma aplicação web;

2. FUNDAMENTAÇÃO TEÓRICA

Antes de se realizar a apresentação de como funciona a arquitetura Flux, seus elementos e características, é necessário um entendimento sobre o funcionamento da internet e a importância da arquitetura, para se desenvolver softwares de qualidade, seja ele com seu uso na internet ou não.

2.1 O FUNCIONAMENTO DA WEB

2.1.1 História

Existem diversas menções, artigos e livros sobre o surgimento da internet, e todos eles apontam a origem da internet ter ocorrido durante a Guerra Fria. Neste documento, será apresentado um breve resumo sobre este surgimento e sua evolução, ressaltando apenas fatores e elementos importantes para a compreensão do tema deste trabalho.

De acordo com o Techtudo (2016), a internet foi criada através de pesquisas militares durante a Guerra Fria. Durante esta época, Estados Unidos e a União das Repúblicas Socialistas Soviéticas, possuíam grande controle e influência no mundo, sendo assim, qualquer nova ferramenta ou meio, poderia ajudar algum destes países durante este período de guerras.

Como o mundo havia acabado de presenciar a Segunda Guerra Mundial, que teve seu fim em 1945, ambos os países sabiam da necessidade dos meios de comunicação. Techtudo (2006) relata que os Estados Unidos temiam por um ataque russo às suas bases militares. Caso acontecesse, o ataque poderia tornar público informações sigilosas, o que o tornaria vulnerável. Foi criado então um *modelo* de troca e compartilhamento de informações, que se torna possível a descentralização das mensagens, ou seja, caso o Pentágono fosse atacado, as informações guardadas naquele

local poderiam ser acessadas de outro local. Foi criada então uma rede para o compartilhamento destas mensagens, a ARPANET. (TECHTUDO, 2016)

Surgiu-se então uma necessidade de se padronizar a forma de envio e recebimento destas mensagens dentro da rede. Segundo o Techtudo (2016), a primeira descrição do protocolo TCP foi feita no ano de 1973, por Net Vinton Cerf e Bob Kahn. A utilização do verbete Internet, para uma rede TCP/IP em nível global, teve início em dezembro de 1974, com a divulgação da primeira especificação completa do TCP, assinada por Vinton Cerf, Yogen Dalal e Carl Sunshine, na Universidade de Stanford. O protocolo TCP, por ser a base da comunicação na internet, será melhor definido na sessão 2.1.2 deste documento.

Ainda de acordo com o Techtudo (2016), Tim Berners-Lee, em 1992, criou a World Wide Web. Esta rede propôs a criação dos hipertextos, tornando possível que várias pessoas trabalhassem juntas, visualizando os mesmos documentos. Ainda nos anos 90, a empresa Netscape criou o protocolo HTTPS, que permitia o envio de dados criptografados na web, e também surgiram os primeiros navegadores da internet (conhecidos como *browsers*), como o Mosaic e o Netscape. Com a popularização desta rede e do surgimento dos *browsers*, as pessoas começaram a ter seus próprios computadores pessoais e acessar a esta rede de sua própria casa. Na década de 90, começaram a surgir grandes portais, como AOL e Yahoo, serviços de mensagens instantâneos, como o ICQ e os serviços de e-mail gratuitos, como o Hotmail, e serviços buscadores, como o Google e Cadê.

2.1.2 Estrutura e Funcionamento

Para se ter um mínimo entendimento sobre o ecossistema e tecnologias envolvidas no desenvolvimento de aplicações web, é preciso compreender alguns conceitos e princípios de funcionamento de rede de computadores, principalmente da arquitetura do tipo cliente-servidor. Nesta sessão, será apresentada uma breve abordagem sobre este conteúdo, mas que servirá de base para o entendimento do conteúdo apresentado neste

trabalho e principalmente, compreender as diferenças entre *client-side* (ou front-end) e *server-side* (ou back-end).

Para Santos (2016), quando um usuário, através de seu navegador de internet, solicita por alguma determinada página na internet, ocorre um processo de busca por aquele conteúdo, e ao ser localizado na rede, este conteúdo é retornado ao usuário, sendo exibido em seu navegador.

Este processo de requisição e resposta por meio do protocolo HTTP acontece através da conexão estabelecida entre o cliente e o servidor por meio do protocolo TCP. As mensagens de requisição e resposta geradas pelo navegador e servidor web são quebradas em pacotes e enviadas através da rede com toda a infraestrutura que o TCP oferece. Esta abordagem que os sites e aplicações web utilizam é conhecida como arquitetura cliente-servidor. " (SANTOS, 2016)

Assim, pode-se ter um melhor entendimento ao se visualizar o processo de pedido e resposta de uma página web, conforme demonstra a Figura 1 abaixo:

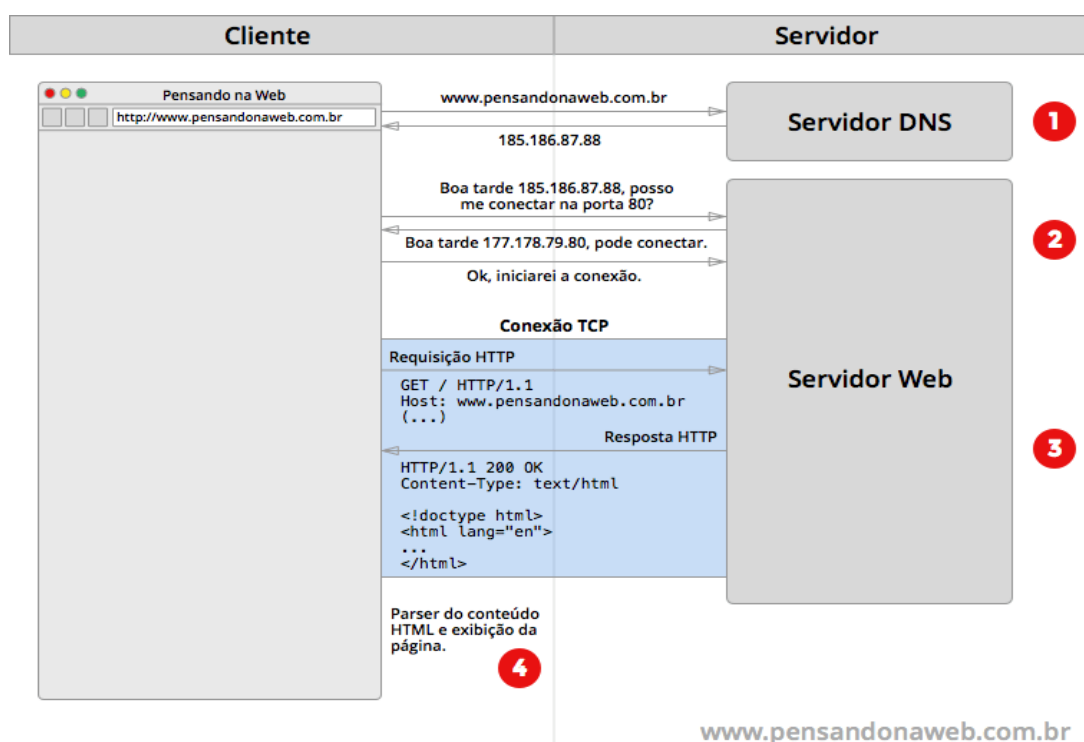


Figura 1 – Processo de pedido e resposta de uma página web

Fonte: (<http://pensandonaweb.com.br/como-funciona-a-internet-e-a-world-wide-web/>)

Na foto acima, é possível notar duas sessões bem distintas e que formam o funcionamento de toda a internet, bem como a interação dos usuários com ela. De um lado, temos o *client-side*, também conhecido como front-end. Nesta sessão é onde se localizam os navegadores de internet, que solicitam as páginas a serem acessadas. Todo o conteúdo que será exibido no *browser*, provém do *server-side*, ou *back-end*. Neste trabalho, todo o foco e conteúdo será destinado ao *client-side*, que é responsável por apresentar as páginas web e os seus conteúdos ao usuário³.

2.1.3 Tecnologias utilizadas e evolução das aplicações *Client-side*

Conforme apresentado na sessão anterior, todos os dados que são exibidos para o usuário em seu navegador, após solicitar por uma determinada página na internet, é destinado ao *client-side*, ou, *front-end*. Para se desenvolver páginas e aplicações web, fazemos uso das tecnologias HTML, CSS e Javascript. (<http://pensandonaweb.com.br/como-funciona-a-internet-e-a-world-wide-web/>)

O HTML fica responsável por montar a estrutura da página, bem como o seu conteúdo. O CSS é a tecnologia responsável por adicionar estilos visuais aos conteúdos e a estrutura, e o Javascript é responsável pelo comportamento dos elementos e da estrutura da aplicação. ((<http://pensandonaweb.com.br/como-funciona-a-internet-e-a-world-wide-web/>))

Com a evolução destas tecnologias, as aplicações web ganharam mais espaço e visibilidade, do que apenas no cenário dos computadores pessoais. A mesma aplicação que antes era construída pensada apenas para usos em computadores pessoais, agora também é acessada nos dispositivos móveis. Além disso, é possível utiliza-las, juntamente com outras ferramentas, para construir aplicativos para smartphone, utilizando tecnologias de desenvolvimento web. (<http://pensandonaweb.com.br/como-funciona-a-internet-e-a-world-wide-web/>)

³ Disponível em <http://pensandonaweb.com.br/como-funciona-a-internet-e-a-world-wide-web/>

Um dos padrões mais consagrados existentes em termos de arquitetura é o *modelo MVC*, ou *Model-View-Controller*. Elaborado por Trygve Reenskaug, este padrão sempre pregou o princípio da separação de responsabilidades, ou seja, cada camada era responsável unicamente por uma determinada parte do sistema. As camadas eram compostas nas seguintes partes:

2.2. ARQUITETURA DE PROJETOS DE SOFTWARE

Desenvolver aplicações capazes de gerar o valor esperado para os usuários é sempre uma tarefa complexa a ser feita. De acordo com Sommerville (2003), “mesmo os sistemas de software mais simples possuem uma alta taxa de complexidade”. Portanto, é necessário empregar os princípios de engenharia de software, durante a etapa de desenvolvimento do projeto, com o objetivo de se entregar software com qualidade.

Sommerville (2003) traz que a busca por se obter maior qualidade no desenvolvimento de aplicações trouxe como consequência a criação de diversos padrões e técnicas diferentes, que contribuem na criação de softwares. Apesar da grande quantidade destes padrões, todos estes padrões buscam o desenvolvimento de soluções e projetos de software com o menor número possível de erros, facilitar o processo de manutenção e evolução do código, de uma forma mais fácil e organizada, e que atenda as necessidades do usuário.

Devido a quantidade de fatores e elementos envolvidos durante o processo de construção de um software, torna-se bastante difícil criar alguma solução que não possua erros. Contudo, o emprego das técnicas e boas práticas de engenharia de software, fundamentais na construção de qualquer projeto de software, utilizar-se padrões de projeto ou padrões arquiteturais, nos permite o reaproveitamento do código e torna mais fácil a evolução da aplicação.

Para Sommerville (2003) durante o desenvolvimento de um software, são inúmeros os fatores determinantes para uma entrega de qualidade: um deles, é o cuidado e a atenção quanto a arquitetura. A arquitetura de software é o que nos permite criar toda a estrutura necessária para o desenvolvimento de nossa aplicação. É esta estrutura que nos orienta durante toda a etapa de desenvolvimento e criação de um software.

De acordo com Baptistela (2016), no processo de confecção da aplicação, caso a equipe de desenvolvedores e projetistas do software não optarem por escolher algum tipo de organização sobre a estrutura de código a ser criada, mantendo toda a parte de regras de negócio, regras de apresentação dos dados e funcionamento da aplicação, acrescido da apresentação dos dados da aplicação ao usuário, a quantidade de problemas que esta solução pode conter será muito grande, visto que qualquer alteração feita nesta estrutura pode comprometer outras partes em funcionamento do sistema.

Atualmente em nosso cotidiano percebemos algumas empresas de desenvolvimento de software entregando aos seus clientes softwares gigantescos, e de alta complexidade, e dessa maneira alguns sistemas apresentam fracas estruturas, deixando de atender as expectativas dos seus clientes, há ainda outros casos de empresas que atrasam a entrega do software tendo que renegociar os prazos pagando pelo não comprimento da entrega do produto, parte destas empresas desenvolvedoras de software não utiliza nenhum tipo de padrões ou arquitetura no planejamento e execução do software. (ADRIANO JOSÉ BAPTISTELLA, 2016, p.2)⁴

Na Figura 2, pode-se exemplificar, de acordo com Pressman (1995), a evolução em desenvolvimento de software.

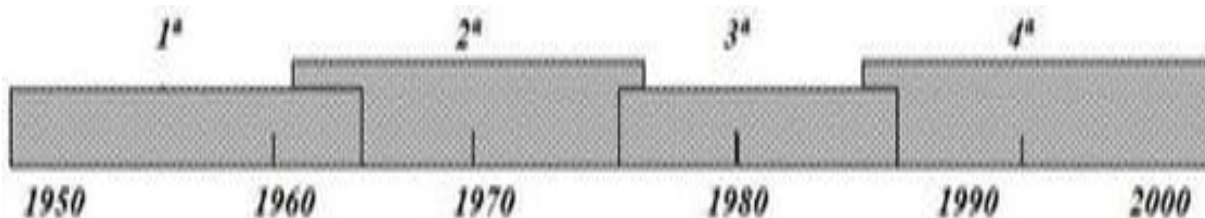


Figura2 - A evolução do software. (PRESSMAN, p. 5, 1995).

⁴ Disponível em: <http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx>

Nos primeiros anos, o hardware sofria grandes mudanças e o software era tratado como algo secundário, logo, o desenvolvimento de projetos de software era feito sem organização ou estrutura. Contudo, a medida que a interação com o usuário foi intensificada e quantidade de ações realizadas eram maiores, o desenvolvimento de software, além de ser visto com certa importância, a complexidade de desenvolvimento ficava cada vez maior, logo, percebe-se a importância da adoção de algum *modelo* ou padrão de arquitetura de aplicações.

2.2.1 O padrão de arquitetura MVC

Um dos tipos de padrões de arquitetura existentes mais consagrados e utilizados em termos de desenvolvimento de aplicações, principalmente em soluções web, é o *modelo* MVC, ou *Model-View-Controller*. Segundo Fowler (2003), o padrão foi criado por Trygve Reenskaug, ao final da década de 1970, para ser utilizado junto a linguagem Smaltalk, tendo fundamental importância nos elementos e interfaces de interação com o usuário (também conhecidos como *User Interface*) e no pensamento sobre o design da interface do sistema, com o usuário.

Para Fowler (2003), o gerenciamento das interfaces de uma aplicação com o usuário é uma tarefa bastante complexa para os programadores. A quantidade de dados a ser apresentado ao usuário, bem como a forma como ele irá interagir com a aplicação e utilizar estes dados, são importantes elementos que garantem uma boa experiência do usuário, durante o uso do sistema. Sendo assim, esta é uma parte vital para o sucesso de qualquer projeto de software. Ainda de acordo com Fowler (2003), as interfaces gráficas do usuário (também conhecidas como *Graphical User Interface*) não eram comuns nos anos 70. Os *modelos Forms and Controls* e *SmartUi*, que até então era a forma utilizada para tratar a parte do sistema responsável pela GUI, nem sempre resolviam este problema de uma forma

ideal. O MVC então foi uma das primeiras tentativas de se fazer, de uma forma mais séria o tratamento com as interfaces de usuário, em qualquer tipo de escala.

A medida que uma aplicação cresce, a quantidade de informações apresentadas para os usuários e as possibilidades de interação com o sistema ficam maiores. De acordo com Fowler (2003), conseqüentemente, esta aplicação começa a ter uma complexidade maior, fazendo-se necessário a adoção de algum padrão ou prática para a arquitetura do projeto, de forma que seja possível realizar possíveis mudanças no projeto, sem comprometer o funcionamento das demais partes do sistema e garantindo a interação do usuário com o sistema.

Segundo Trygve (1979), o principal objetivo do MVC é preencher a lacuna entre o *modelo* mental do usuário humano e o *modelo* digital existente no computador. O MVC foi criado como uma solução geral para o problema da interação dos usuários que controlam e utilizam um conjunto de dados grande e complexo, de uma determinada aplicação.

2.2.1.1 Funcionamento

O *modelo* apresentado por Trygve (1979), o MVC, ou *Model-View-Controller*, possui três camadas bem definidas. Abaixo segue a definição de cada uma das camadas:

- *Model*: Segundo Trygve (1979), os *modelos* representam o conhecimento. Um *modelo* poderia ser um único objeto (bastante desinteressante), ou poderia ser alguma estrutura de objetos. Deve haver uma correspondência um-para-um entre o *modelo* e suas partes, por um lado, e o mundo representado como percebido pelo proprietário do *modelo*, por outro lado.

Conforme descrito acima, o *Model* é a camada que possui todas as regras de negócio da aplicação, ou seja, aonde devem ficar todo o conjunto de regras de acesso aos dados da aplicação, representando assim às informações (ou dados) da aplicação. Para o autor, ele não deve ter conhecimento de quais

serão as interfaces que serão atualizadas (as *views*), ele apenas deve possuir acesso aos dados da aplicação e fornecer meios para que o *Controller* tenha acesso a eles.

- *View*: De acordo com Trygve (1979), uma visão é uma representação visual de seu *modelo* (*model*). Realçaria normalmente determinados atributos do *modelo* e suprimiria outros. É, portanto, atuando como um filtro de apresentação. Também pode atualizar o *modelo* enviando mensagens apropriadas. Ela terá, portanto, que conhecer a semântica dos atributos do *modelo* que representa.

Segundo a descrição apresentada, a *View* é a camada de apresentação, é quem deverá exibir os dados da camada *Model*, ao usuário final. É esta camada que possibilita a interação do usuário com todo o sistema, possibilitando a entrada e a saída de dados, permitindo a visualização das respostas geradas pelo sistema. A *View* deve ser reflexo do estado do *modelo*, então caso ocorra alguma mudança nos dados do *modelo*, este deverá notificar as suas respectivas *views* de alguma forma, para que elas possam exibir os novos dados do *modelo*.

As *views* não devem conter códigos relacionados à lógica de negócios, elas somente devem se preocupar com a apresentação dos dados contidos no *model*. Nas aplicações web, esta camada é representada utilizando a linguagem de marcação HTML, que é interpretado pelo browser e exibido ao usuário.

- *Controller*: De acordo com Trygve (1979), um *controller* é a ligação entre um utilizador e o sistema. Fornece ao usuário entrada, organizando *views* relevantes para se apresentarem em locais apropriados na tela. Fornece meios para a saída do usuário, apresentando ao usuário com menus ou outros meios de dar comandos e dados.

Assim, a definição apresentada acima, o *Controller* é a camada de lógica, é quem faz a ligação da camada *Model*, com a camada de *View*. É responsável por fazer as manipulações dos dados da camada *model* e enviá-los para a camada *view*. Sendo assim, ele gerencia o envio de requisições feitas entre a *view* e o *model*. De acordo Trygve Reenskaug (1979), com O *controller* define todo o comportamento e funcionamento da aplicação, quem

interpreta as ações (cliques de botão ou link, seleção de menu, envio de dados em um formulário) feitas pelos usuários da aplicação.

O *controller* deverá mapear as ações do sistema, já que a medida que o usuário faz as suas solicitações (ou requests) em uma *view*, o *controller* comunica-se com o *model* correto, atualizando então os dados na *view* que o usuário fez a solicitação. Abaixo, observa-se de acordo com a figura 3 uma visão de todo o funcionamento do *modelo* MVC.

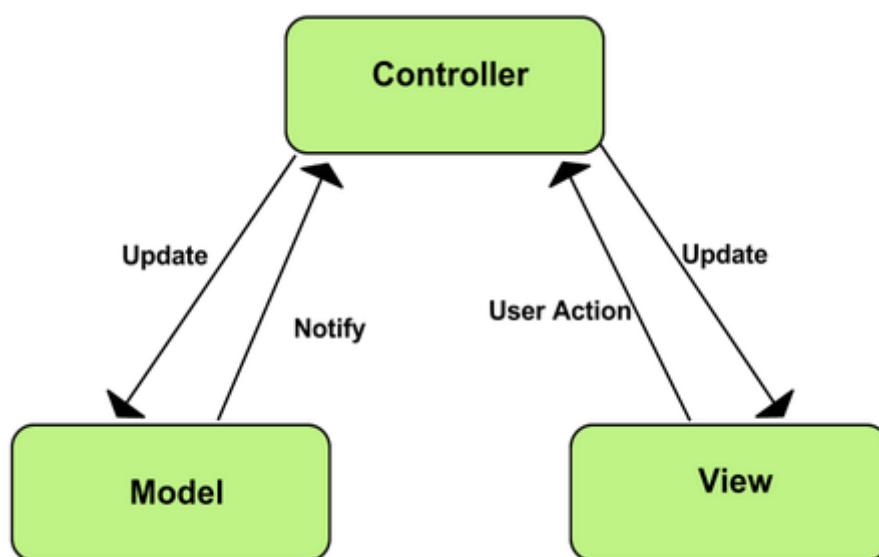


Figura 3: Diagrama do *modelo* MVC, (https://developer.chrome.com/apps/app_frameworks)

2.2.1.2 Características

De acordo com Fowler (2003), o padrão MVC sempre pregou o princípio da separação de responsabilidades, ou seja, cada camada deste padrão era responsável unicamente por uma determinada parte do sistema, de forma que cada uma poderia ser mantida de forma independente.

Fowler (2003) traz que como consequência, este princípio possibilitava a separação das regras de negócio, com a apresentação destes dados ao usuário. Isto possibilita a criação de várias outras *views*, sem afetar outras partes do sistema, o que permite que o código criado seja mais facilmente reutilizável e fácil de ser testado. Possibilita também a adoção de

diversas bibliotecas de código e outras tecnologias, que possam promover maior interação dos usuários com a aplicação.

Este princípio também permitiu que durante o processo de criação de um software, que as equipes trabalhassem de forma paralela, sem que uma equipe precise esperar o resultado por outra. Segundo Fowler (2003), como as camadas do *modelo* atuam de forma independente, cada equipe pode trabalhar em cada uma delas, de forma paralela. Isto também facilitou o processo de manutenção das aplicações, já que a alteração em uma determinada camada não afetaria o funcionamento de outra. Logo, o software fica mais fácil de ser testado, pois cada componente da interface, ou cada camada, podem ser testados separadamente. Caso seja necessário, podemos utilizar o conceito de cópia de objetos (conhecido tecnicamente como *mock*) onde é possível simular o comportamento destes outros objetos.

2.2.1.3 Variações do MVC

Ao longo dos anos, as aplicações web foram ficando mais complexas e possibilitavam aos usuários, maiores interações. Sendo assim, iniciou-se um processo para se obter melhores utilizações do *modelo* MVC. Devido a sua facilidade de implementação e adaptação, surgiram diversos outros *modelos*, inspirados no MVC. Assim, o presente trabalho necessita abordar ainda os *modelos* MVP e MVVM, que foram os padrões mais utilizados pelos frameworks javascript, para a criação de aplicações *client-side*. Segundo Addy Osmani (2015), embora alguns frameworks javascript tenham optado por uma implementação com algumas variações do *modelo* MVC, elas ainda utilizam alguns conceitos fundamentais criados pelo *modelo*.

2.2.1.3.1 MVP

Para Addy Osmani (2015), o padrão de arquitetura MVP, também conhecido como *Model-view-presenter*, é uma derivação do padrão MVC, com foco em melhorar a lógica de apresentação. Foi criado em uma empresa chamada Taligent no início de 1990, enquanto eles estavam trabalhando em um *modelo* para um ambiente C++ CommonPoint. A camada Presenter, que é a grande diferença este *modelo* para o MVC, é um componente que contém a lógica de negócios da interface de usuário. As chamadas das *views* são delegadas ao presenter, e por sua vez, observa os *model* e atualiza as *views* caso os *models* mudam, fazendo o vínculo entre eles.

Através do presenter, que se comunica bidirecionalmente com as outras camadas, evita que o *Model* comunique-se diretamente com o *View*. Para Addy Osmani (2015), o *modelo* MVP permite o desacoplamento das funções, tornando a arquitetura mais modular. Existe ainda uma outra abordagem do MVP, onde a *view* consegue se comunicar com a *model* diretamente. Isso garante que, em casos de aplicações em interfaces mais complexas, o acesso da *view* diretamente torna o acesso aos dados mais fácil. Uma das formas de se utilizar esta arquitetura, em aplicações *client-side*, é através da biblioteca Riot.js (<http://riotjs.com/>). Segue exemplo das duas abordagens do *modelo* MVP, conforme a figura 4 abaixo:

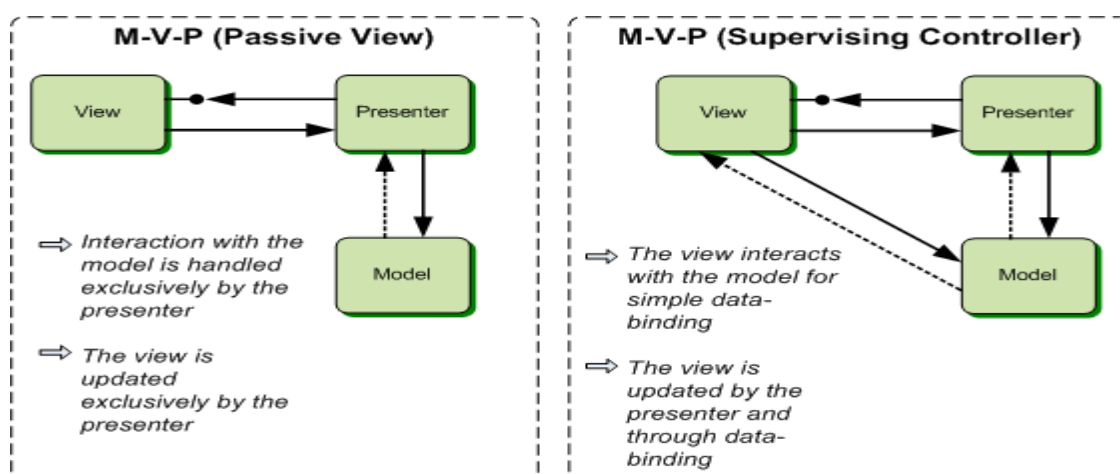


Figura4: Diagrama de funcionamento do *modelo* MVP – <https://sites.google.com/site/akstechtalks/tech-topics/Architecture/patterns/architectural-patterns/presentation-patterns/web-presentation-patterns/model-view-presenter-mvp-pattern>

2.2.1.3.2 MVVM

O padrão MVVM, ou *Model-view-viewmodel*, de acordo com Addy Osmani (2015), é um padrão arquitetural baseado no MVC, sendo uma evolução do MVP, com o objetivo de separar, de forma mais clara, o desenvolvimento de interfaces de usuário, das regras de negócio e do comportamento da aplicação. MVVM foi originalmente definido pela Microsoft para uso com Windows Presentation Foundation (WPF) e Silverlight, sendo anunciado em 2005 por John Grossman.

Addy Osmani (2015) expõe que na arquitetura MVVM, o *ViewModel* não tem conhecimento do que ocorre na *view*, mas a *view* possui este conhecimento do que ocorre no *ViewModel*. Isso se deve ao fato de as marcações de vínculo para o *ViewModel* serem feitas na *View*, através da linguagem de marcação HTML. No que se diz respeito ao *Model*, ambas as camadas conseguem enxergar o estado atual do *Model*. Este *modelo* incrementa propriedades e operações ao *Model*, de forma a atender as necessidades da *View*, criando um novo *modelo* para a visualização das informações contidas no *Model*.

Para utilizar esta arquitetura, em aplicações *client-side*, podemos utilizar a biblioteca KnockoutJS⁵. Segue abaixo a figura 5 exemplificando as duas abordagens do *modelo* MVVM.

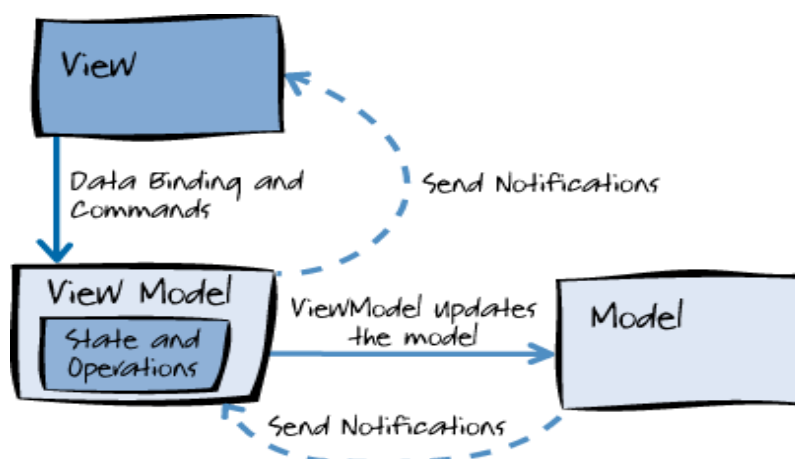


Figura5– Diagrama de funcionamento do Modelo MVVM.

Fonte - <https://msdn.microsoft.com/en-us/library/hh848246.aspx>

⁵ Disponível em: <http://knockoutjs.com/>.

2.2.1.4 MVC em aplicações *client-side*

De acordo com Amir Salihefendic (2015), com a evolução das linguagens e tecnologias utilizadas para o desenvolvimento de aplicações *client-side*, percebe-se a necessidade de adicionar e de se utilizar um *modelo* de arquitetura para ela também, já que estas aplicações cresceram de tal maneira que não era apenas uma questão de exibir dados do *server-side*, e sim, oferecer uma total e grande experiência na utilização da aplicação. Para isso, sem dúvida alguma, o *modelo* MVC foi e ainda é o *modelo* de arquitetura mais utilizado.

Contudo, de acordo com PAUL COWAN (2013), “as aplicações *client-side*, baseadas no *modelo* MVC, se comportam completamente diferente de aplicações MVC baseadas em *Server Side MVC*”, uma vez que as aplicações *server-side* seguem mais à risca, o *modelo* tradicional do MVC. Em uma aplicação *server-side*, baseada no *modelo* MVC, existe um único ponto de entrada em toda a aplicação. Quando é feita uma requisição pelo navegador, o servidor web ira determinar qual é a rota solicitada por aquela requisição, acionando o *controller* responsável por realizar a ação solicitada pelo pedido. O *controller* se encarrega de acessar os dados contidos no *Model* e retorna este conteúdo ao servidor web, que por sua vez retorna estes dados ao browser do usuário, com os dados que ele havia solicitado. Devido a própria estrutura do protocolo HTTP, em uma requisição, existe um conjunto específico de dados que podem ser enviados para o servidor web. Após a requisição ser processada, a aplicação pode retornar um conjunto de dados para o browser, que podem ser do tipo html, json, xml, ou meramente texto puro. (PAUL COWAN, 2013)

No lado do cliente (browser), não existem restrições como quais os objetos de dados podem continuar existindo, enquanto a sessão do navegador não for fechada. Segundo Paul Cowan (2013), “Um *model* pode notificar a *view* de quaisquer alterações, através do padrão Observer”. Também é importante ressaltar que existem várias ações realizadas na *view* que podem atualizar o

Model, como uma entrada de dados em um formulário, uma requisição AJAX que pode atualizar os dados do *model* e a mudança em campos de formulários faz com que alguns frameworks javascript usem o evento *onchange* para atualizar os valores do *Model*.

O autor também afirma que uma das principais e famosas tecnologias utilizadas em aplicações *client-side*, que permite utilizar a arquitetura MVC é o AngularJS (<https://angularjs.org/>). Existem muitas outras ferramentas, como o EmberJS (<http://emberjs.com/>) e o BackboneJS (<http://backbonejs.org/>), mas o Angular tem sido uma escolha bastante comum entre os desenvolvedores que adotam este tipo de arquitetura em suas aplicações. Será exibido alguns trechos de código de uma aplicação MVC no Estudo de Caso.

2.3 O SURGIMENTO DO FLUX

Em 2014, na conferência F8, o time de Engenharia do Facebook apresentava algumas novidades e tecnologias que eles haviam utilizando. Entre elas, a engenheira de software Jing Chen, que trabalha no facebook, apresentou um novo *modelo* de arquitetura que o facebook estava utilizando no front-end de suas aplicações⁶.

Jing Chen iniciou sua apresentação exaltando como o padrão MVC funciona perfeitamente bem em contextos e aplicações pequenas. Porém, quando esta aplicação começa a evoluir, devido a própria estrutura em camadas do *modelo*, quando novas camadas começam a surgir, cria-se uma grande dificuldade no entendimento de como o software deveria funcionar⁷.

De acordo com Mark Richards (2015), o *modelo* de camadas gera um forte acoplamento, ou dependência, entre elas. Sendo assim, fica muito difícil escalar este tipo de aplicação, ou seja, a medida que o sistema cresce, o numero de camadas tende a aumentar, conseqüentemente o elo entre elas fica

⁶ Disponível em <https://facebook.github.io/flux/docs/overview.html>.

⁷ Disponível em <https://facebook.github.io/flux/docs/overview.html>

mais forte. Em uma estrutura deste tipo, torna-se bastante complexo o entendimento de toda a estrutura, para se acrescentar novas funcionalidades na aplicação. Assim, pode-se observar através da Figura 6 a complexidade de uma aplicação feita no padrão MVC.

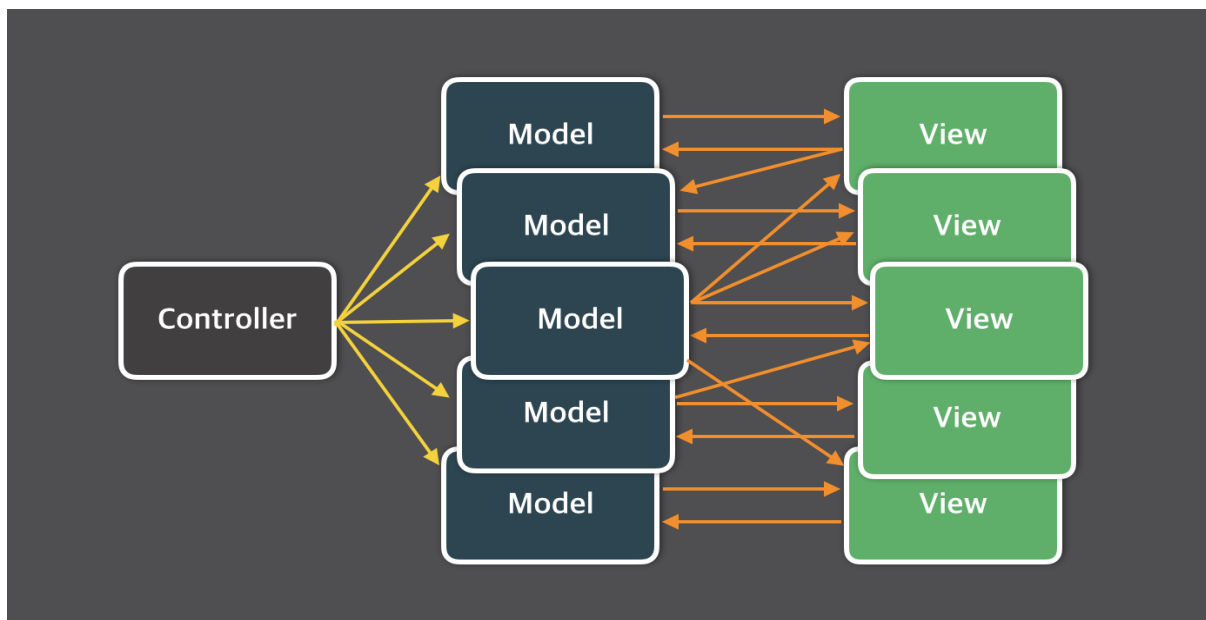


Figura 6: Complexidade de uma aplicação feita no padrão MVC
Fonte: <https://facebook.github.io/flux/docs/overview.html#content>

No contexto acima, fica evidenciado o como é difícil manter o controle e a clareza de uma aplicação, quanto utilizamos este tipo de padrão. Uma única ação pode resultar no acesso a vários *Models*, que por sua vez pode interagir com diversas *Views* diferentes.

Assim, faz-se necessário exemplificar acerca da complexidade de uma aplicação a medida que ela cresce, conforme se demonstra na figura 7 abaixo:

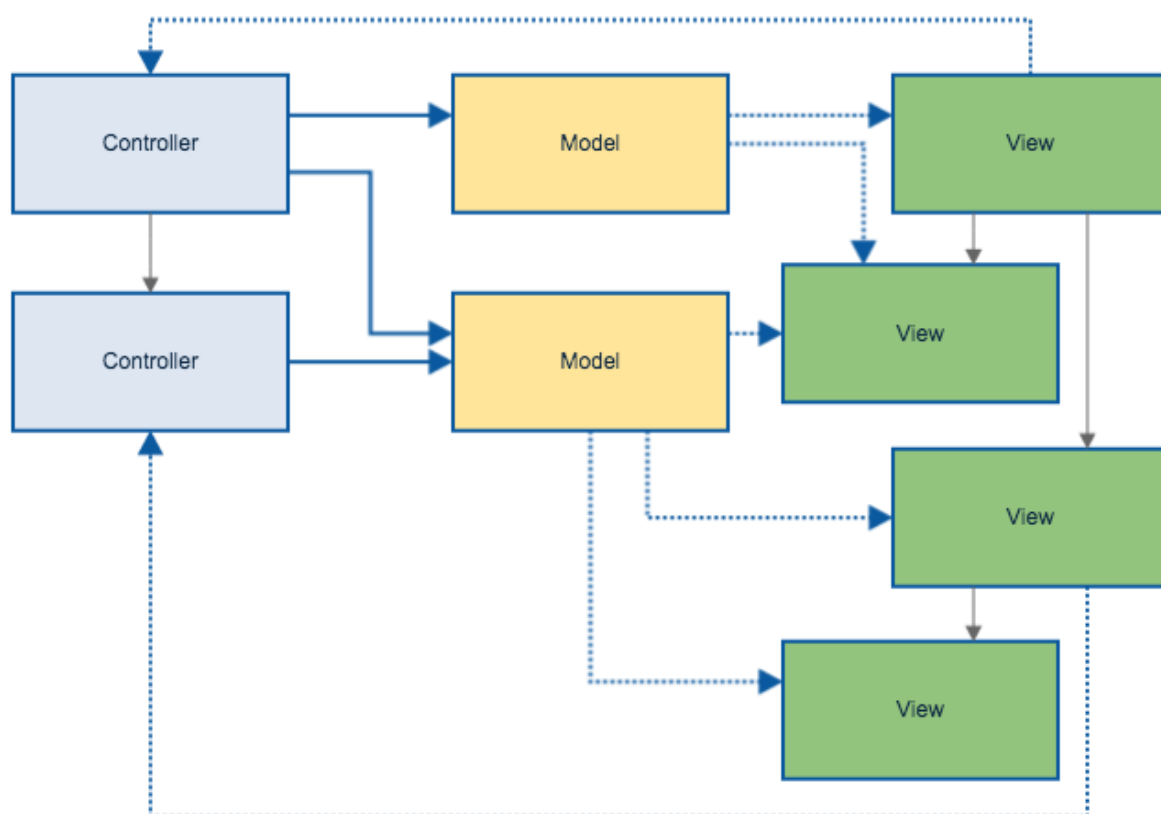


Figura7 – Uma nova visão sobre a complexidade de uma aplicação a medida que ela cresce.
 Fonte - <http://fluxxor.com/what-is-flux.html>

No contexto acima, ao adicionar algumas *views* ao *modelo*, um novo *controller* e um novo *modelo*, o gráfico de dependência entre as camadas já é mais denso, logo é mais complicado de se visualizar o fluxo de dados da aplicação. Quando o usuário interage com a *view*, o fluxo e a quantidade de códigos que podem ser executados são difíceis de serem mapeados, o que gera uma enorme dificuldade em se mapear as ações do sistema e a identificação de possíveis problemas na aplicação. Em um cenário desfavorável, uma interação do usuário irá realizar diversas atualizações, que por sua vez podem desencadear outras atualizações e ações dentro da aplicação, criando um efeito de cascata dentro da estrutura do sistema, o que torna ainda mais complexo identificar possíveis problemas.

Segundo Rodrigo Rebouças (2016), a utilização da arquitetura MVC pode trazer alguns problemas para a aplicação: “Se tivermos muitas visões e o *modelo* for atualizado com muita frequência, a performance do sistema pode ser abalada”; “Se o padrão não for implementado com cuidado, podemos ter casos como o envio de atualizações para visões que estão minimizadas ou fora

do campo de visão do usuário”; “Ineficiência: uma visão pode ter que fazer inúmeras chamadas ao *modelo*, dependendo de sua interface.”

Todo este ciclo cria um enorme problema para se compreender o real funcionamento do software. Sequências de repetições são criadas para se entender todo o funcionamento de uma única ação. E segundo Jing, “esta complexidade impedia o Facebook de criar novas features de uma forma mais rápida, e com uma qualidade alta”⁸.

Sendo assim, eles apostaram em um novo modelo de arquitetura, que não tivesse as limitações apresentadas por ela, e que o fluxo de dados fosse único. Este *modelo* foi chamado de Flux. Que abaixo se exemplifica pela figura 8:

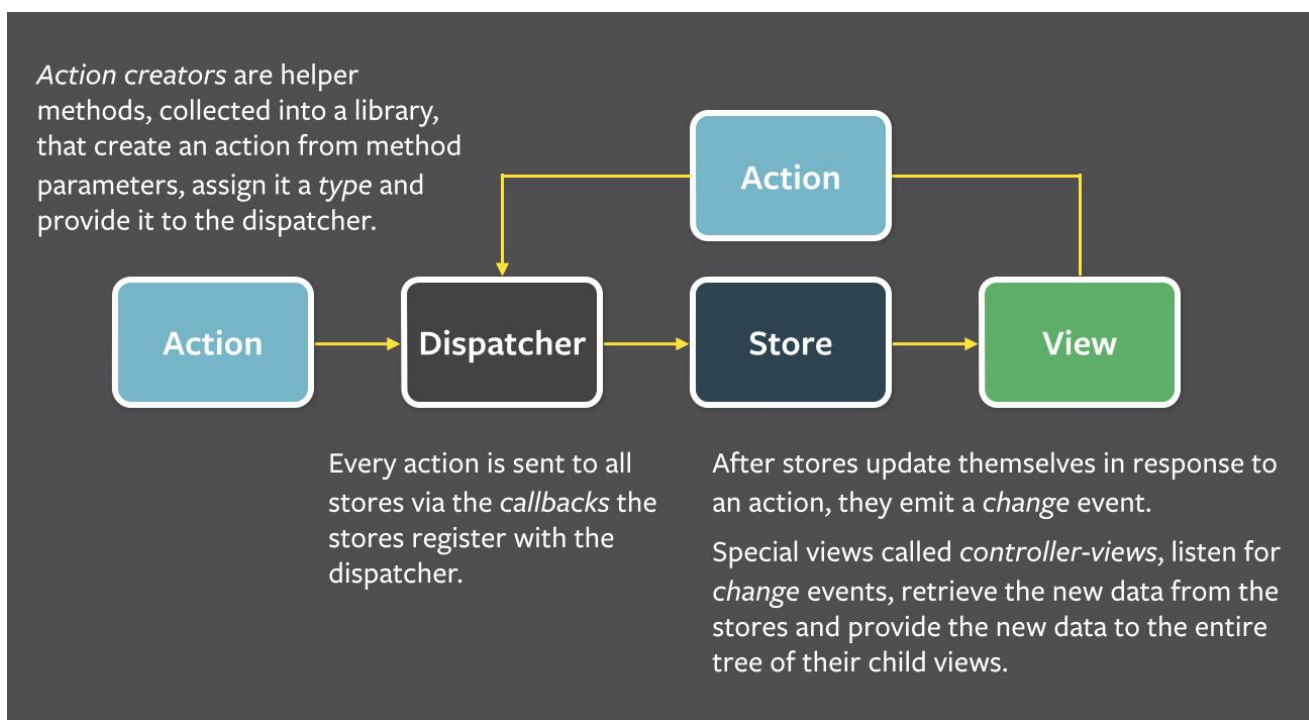


Figura 8: Diagrama do funcionamento da estrutura Flux

Fonte: <https://facebook.github.io/flux/docs/overview.html#content>

⁸ Disponível em <https://facebook.github.io/flux/docs/overview.html>.

2.4 A ARQUITETURA FLUX

Toda a estrutura e fluxo de funcionamento da arquitetura flux consiste em criar um único caminho de dados, eliminando a bi-direcionalidade que existe nas implementações do padrão MVC nas aplicações *client-side*. Além deste único fluxo direcional (traduzido do termo original Single Direction Data Flow), ele possui os elementos: Actions, Dispatchers, Stores e Views. Toda a estrutura de funcionamento será descrita de acordo com a documentação oficial do FLUX.

2.4.1 Fluxo único de dados

Todo e qualquer tipo de informação que for transmitido em um sistema que utilize a arquitetura flux, terá o seu fluxo semelhante ao que se demonstra na Figura 9.

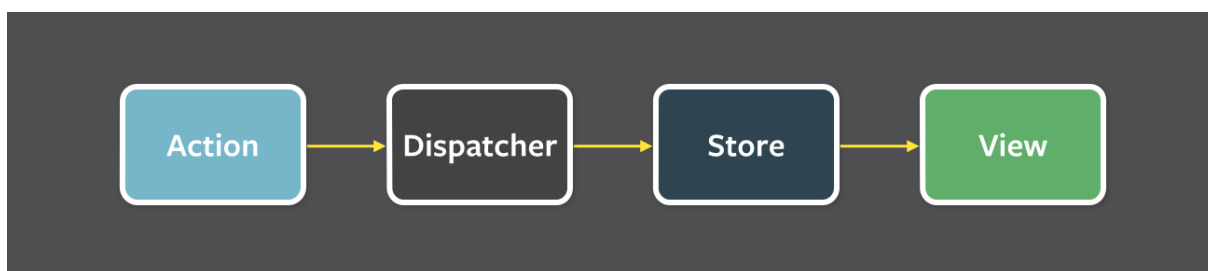


Figura 9: Representação arquitetura flux e o fluxo único de dados
Fonte: <https://facebook.github.io/flux/docs/overview.html#content>

Neste fluxo se concentra toda a parte central e funcionamento desta arquitetura. É este diagrama que o desenvolvedor de software precisa ter em mente, quando for criar toda a estrutura de código de seu sistema. Apesar de fazerem parte da arquitetura, o Dispatcher, a Store e a View são partes totalmente independentes, que por sua vez trabalham com dados de entrada e saídas distintos. As Actions são representadas por objetos (conjunto de dados), que apenas possuem as novas informações a serem salvas pela aplicação.

Este objeto também possui uma propriedade para identificar de qual tipo ele pertence.

As *Views* representam a visualização dos dados dos objetos do sistema, aos usuários. Estas *views* podem fazer com que uma nova ação seja solicitada e propagada através de todo o sistema, respondendo assim as interações do usuário com o sistema. Mesma com essa nova solicitação, o fluxo único de dados se mantém intacto. Conforme se demonstra pela figura 10:

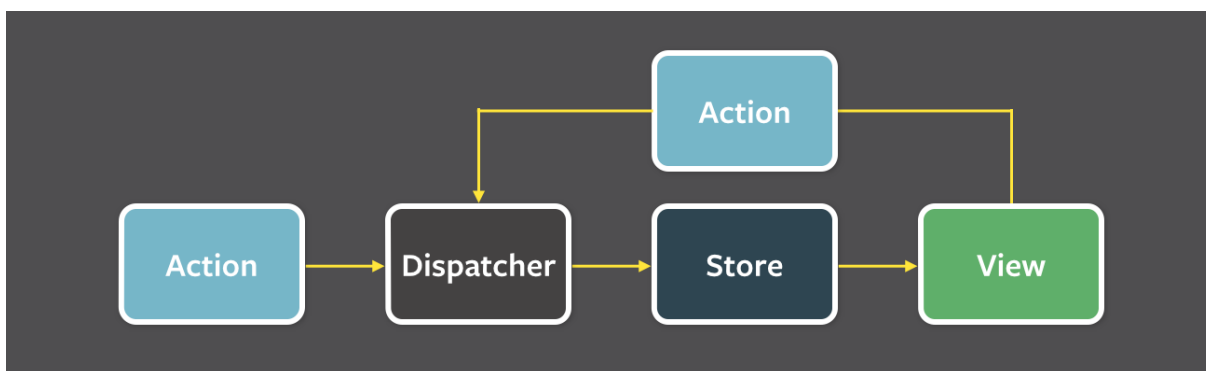


Figura 10: Descritivo de uma nova ação realizada na *View*

Fonte: <https://facebook.github.io/react/docs/flux-overview.html#content>

O fluxo único de dados se mantém, mesmo adicionando a estrutura novas *Stores* e *Views*. O dispatcher simplesmente se encarrega de enviar todas as ações para todas as *Stores* da aplicação, uma vez que o dispatcher não possui nenhum conhecimento sobre as regras de negócio e como atualizar o valor contido nas *Stores*. Cada *Store* é responsável por um domínio da aplicação e somente se atualizam mediante em resposta às actions.

Assim, observa-se as aplicações flux mais complexas, conforme se apresenta abaixo pela figura 11:

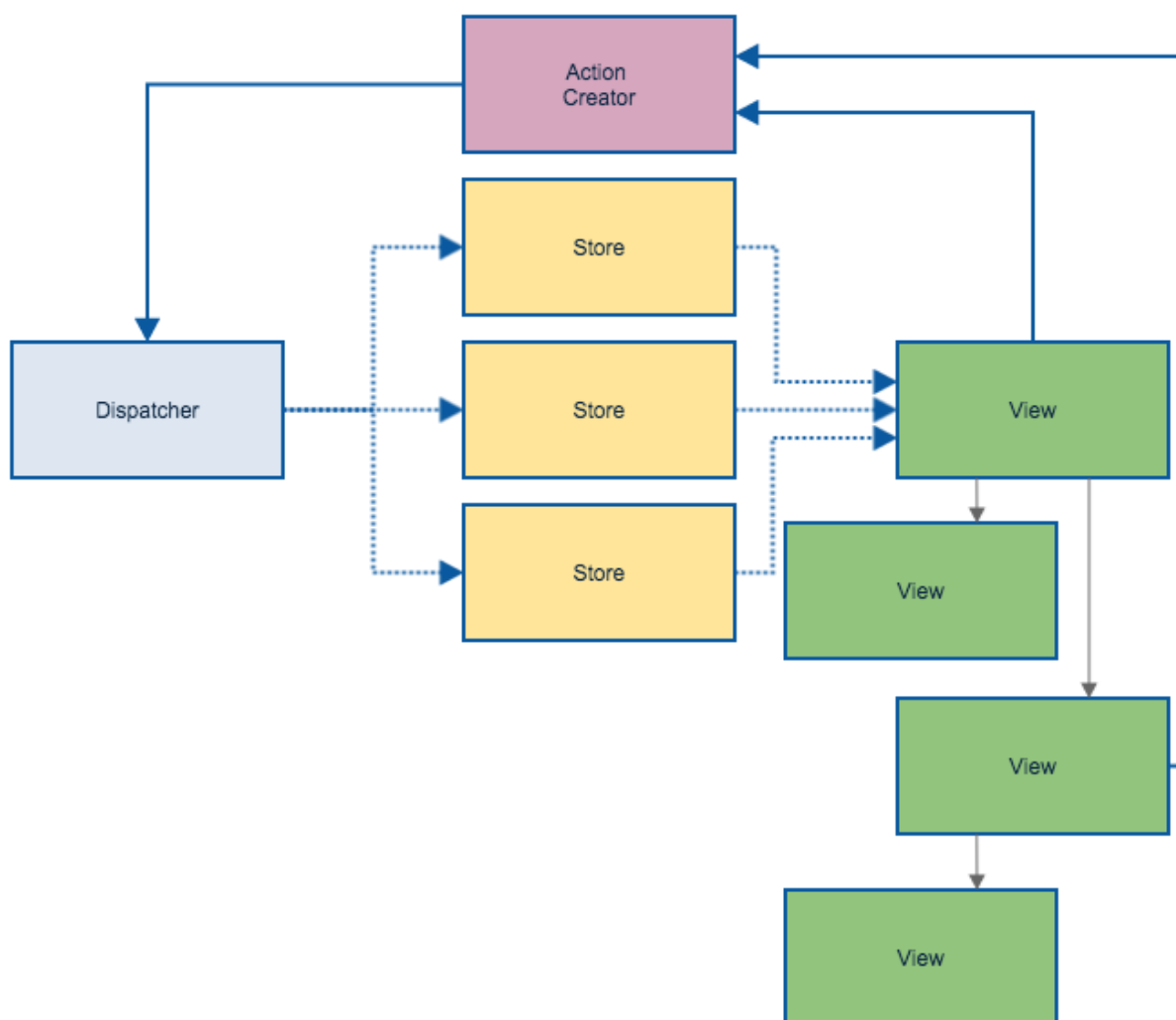


Figura11 – Aplicações flux complexas.
Fonte: <http://fluxxor.com/what-is-flux.html>

2.4.2 Dispatcher

Ele é o concentrador de toda a arquitetura flux, responsável por gerenciar todo o fluxo de dados da aplicação, pois é um mecanismo simples para distribuir as actions para as stores. Como elas não possuem nenhum tipo de inteligência ou conhecimento sobre o funcionamento da aplicação, o

dispatcher consegue registrar algumas rotinas de código que serão executadas após uma determinada informação ser registrada na store⁹.

De acordo com a evolução do software, o dispatcher se torna mais essencial, pois pode ser usado para gerenciar dependências entre as stores, executando as rotinas de código registradas em uma ordem específica. É importante ressaltar que na arquitetura flux, só deverá existir apenas um dispatcher¹⁰.

2.4.3 Actions

O dispatcher oferece um método que nos permite acionar eventos para as stores e salvar novos dados de nossa aplicação. Este conjunto de dados é o que chamado de action. Elas também podem vir de outros lugares, como a parte de *back-end* do software. Isso acontece, por exemplo, durante a inicialização de dados¹¹.

Quando ocorre algum tipo de interação ou ação em alguma interface da aplicação, podemos citar como exemplo um clique de botão de um formulário, ocorre uma ação de dispatch, onde uma action será disparada para toda a aplicação. A action é uma representação de um objeto, na linguagem Javascript, que descreve o que queremos fazer, e os dados que precisamos, ou possuímos, para realizar tal ação. (<http://fluxxor.com/what-is-flux.html>)

2.4.4 Store

As stores contêm o estado (as informações) e a lógica da aplicação. Seu papel é bastante diferente a estrutura *Model* do padrão MVC citado no capítulo 1, uma vez que além de gerenciam o estado de muitos objetos, eles são independentes e não estão presos a uma estrutura física de banco de

⁹ Disponível em <https://facebook.github.io/flux/docs/overview.html>

¹⁰ Disponível em <https://facebook.github.io/flux/docs/overview.html>

¹¹ Disponível em <https://facebook.github.io/flux/docs/overview.html>

dados, nem como a um único tipo de dados. As stores podem conter vários *Models*, do padrão MVC, ou seja, diversos tipos de fontes de dados¹².

As stores são a única parte da arquitetura Flux que tem o conhecimento de como atualizar os objetos, as actions enviadas pela aplicação não tem conhecimento para poder apagar ou alterar itens na aplicação. Depois que elas são atualizadas, eles transmitem um evento declarando que seu estado foi alterado, para que as todas as *views* podem consultar o novo estado das informações e exibi-las para os usuários¹³.

2.4.5 Views

As *views* são todo o conjunto de elementos e estruturas visuais desenhados na página web, que irão apresentar as informações contidas na store, aos usuários do sistema. Existem diversas formas, meios e tecnologias que podem ser utilizadas, mas a ferramenta indicada pelo Facebook, é o React¹⁴.

2.4.5.1 A biblioteca React

Para aumentar o conhecimento sobre a estrutura *view* do ecossistema Flux, precisamos ter um mínimo de conhecimento na biblioteca responsável por este importante aspecto na arquitetura Flux. Segundo a proposta do Facebook, esta parte é gerenciada pela ferramenta React. (<http://fluxxor.com/what-is-flux.html>)

Criada pelo time do Facebook, a biblioteca React permite criar elementos, ou componentes visuais, em páginas web. Ela é baseada nos conceitos de componentes web (Traduzido do termo *web components*) criado

¹² Disponível em <https://facebook.github.io/flux/docs/overview.html>

¹³ Disponível em <https://facebook.github.io/flux/docs/overview.html>

¹⁴ Disponível em <https://facebook.github.io/flux/docs/overview.html>

pelo W3C, o consórcio responsável por gerir e determinar os padrões da internet¹⁵.

Este padrão consiste em agrupar as lógicas de funcionamento, comportamento e estilização de uma forma mais unitária, onde cada bloco da sua aplicação web consegue funcionar de forma única¹⁶.

Além de ser baseado em componentes, e de possuir uma sintaxe simples e declarativa, um das grandes vantagens do React é o uso do Virtual DOM. Este conceito e estrutura permite à ferramenta uma associação dos componentes, com o seu respectivo elemento na página, em uma estrutura de árvore binária. Toda a manipulação é feita nesta estrutura intermediária antes de ser refletida no DOM, o que torna esta ferramenta altamente performática¹⁷.

Abaixo, temos um exemplo de código inicial sobre como criar um elemento web utilizando a biblioteca React.

```
ReactDOM.render(
  <h1>Primeiro exemplo com React</h1>,
  document.getElementById('root') );
```

Fonte: Elaborado pelo próprio autor

Toda a utilização se baseia na chamada principal explícita da biblioteca através da sintaxe ReactDOM. Utilizamos o método render para determinar qual elemento HTML queremos adicionar e aonde queremos. Existem muitas outras funcionalidades e possibilidades que o React proporciona para o desenvolvimento de aplicações *client-side*, contudo, para o entendimento da arquitetura Flux, este pequeno contato é suficiente para prosseguir no entendimento dos problemas abordados neste trabalho.

2.5 CARACTERÍSTICAS

¹⁵ Disponível em <https://facebook.github.io/flux/docs/overview.html>

¹⁶ Disponível em <https://facebook.github.io/flux/docs/overview.html>

¹⁷ Disponível em <https://facebook.github.io/flux/docs/overview.html>

Conforme descrito na sessão anterior sobre todo o funcionamento da arquitetura Flux, é possível detectar algumas características e pontos chave para o entendimento e o sucesso desta arquitetura.

Como as stores se atualizam internamente, em resposta às actions, (ao contrário de serem atualizadas por outra camada externa, como um controller), nenhuma outra parte do sistema precisa saber como modificar o estado da aplicação. Toda a lógica para atualizar o estado está contida dentro da própria store, e uma vez que esta atualização ocorre apenas de forma síncrona, para testar as stores de nossa aplicação, basta colocá-las em um estado inicial, enviando-lhes uma ação e testando para ver se elas estão no estado final desejado.

Como as lojas stores se atualizam em resposta das actions, as actions tendem a ser mais descritivas, ou mais semânticas, uma vez que as actions não sabem como de fato realizar uma determinada ação. Esta melhor descrição das actions facilita o entendimento de todo o fluxo de atividades da aplicação.

Um outro fator importante é que a arquitetura flux não permite enviar uma segunda ação como resultado de um dispatch de uma action. Isso diminui as atualizações em cascata na aplicação, e ajuda a construir o fluxo de possibilidades de interação na aplicação, em torno de actions mais semânticas.

2.6 ADOÇÃO DO FLUX

Desde o seu anúncio oficial, a arquitetura Flux tem sido cada vez mais utilizada e discutida pela comunidade dos desenvolvedores de aplicações client-side. Toda a estrutura da arquitetura, documentação, explicação e exemplos de como utilizar a arquitetura estão disponíveis no repositório oficial do projeto no Github, disponível em <https://github.com/facebook/flux>. O Github é um serviço online dedicado a hospedagem de projetos que utilizam o controle de versão Git. Por ser open-source, facilita a interação da comunidade de desenvolvedores com a arquitetura e seu ecossistema.

Assim, observa-se pela figura 12 a representação do projeto Flux hospedado no Github:

facebook / flux

Unwatch 674 Unstar 12,819 Fork 3,478

Code Issues 20 Pull requests 22 Projects 0 Wiki Pulse Graphs

Application Architecture for Building User Interfaces <http://facebook.github.io/flux/>

352 commits 2 branches 6 releases 90 contributors BSD-3-Clause

Branch: master New pull request Create new file Upload files Find file Clone or download

steven5538 committed with kyldvs update license year to range (#386) Latest commit 07652c2 3 days ago

| | | |
|-----------------|---|---------------|
| dist | Update dist for 3.1 | a month ago |
| docs | removed FluxMapStore | 7 months ago |
| examples | Fix global key var instead of local one in TodoStore test. (#370) | 4 months ago |
| scripts | jest@15 (#377) | 3 months ago |
| src | Allow register and unregister to be called during dispatch | a month ago |
| website | fix the analytics account (#374) | 3 months ago |
| .gitignore | Add flux utils | a year ago |
| .npmignore | clean up flowconfig a bit | 11 months ago |
| .travis.yml | Added travis ci support (#363) | 5 months ago |
| CHANGELOG.md | Allow register and unregister to be called during dispatch | a month ago |
| CONTRIBUTING.md | Update CONTRIBUTING.md | 2 years ago |

Figura 12 – Representação do projeto Flux hospedado no Github
Fonte - <https://github.com/facebook/flux>

Devido a quantidade de estrelas e forks (ramificações onde é possível contribuir com o código do projeto), é possível notar a grande participação da comunidade de desenvolvedores. É notório um interesse por parte dos profissionais quanto a utilização da arquitetura, bem como sobre a sua evolução. E o fato de ser uma tecnologia criada, mantida e utilizada pelo Facebook, é um grande motivo que evidencia a busca dos desenvolvedores para a utilização desta arquitetura em seus projetos.

A medida que a arquitetura foi sendo utilizada, várias bibliotecas foram criadas com o objetivo de auxiliar a utilização da arquitetura Flux, que utilizavam tecnologias e bibliotecas diferentes das que eram apresentadas pelo Flux em sua documentação original. Segue abaixo uma foto do projeto Flux Comparasion, também disponível no Github, que possui exemplos de todas as implementações do Flux, até então criadas, demonstradas pela figura 13:

voronianski / flux-comparison

Watch 86 Star 2,457 Fork 188

Code Issues 8 Pull requests 1 Projects 0 Wiki Pulse Graphs

Practical comparison of different Flux solutions <http://pixelhunter.me/post/110248593059/flux-solutions-compared-by-example>

219 commits 1 branch 0 releases 23 contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

| Commit | Message | Time |
|---------------|--|---------------------------------|
| voronianski | add shopping cart by vuex to readme | Latest commit fca5a7c on May 27 |
| alt | have common libs in root package.json and specific deps in folders | a year ago |
| common | remove unused file | 2 years ago |
| facebook-flux | have common libs in root package.json and specific deps in folders | a year ago |
| flummox | have common libs in root package.json and specific deps in folders | a year ago |
| fluxette | have common libs in root package.json and specific deps in folders | a year ago |
| fluxury | Single quotes! | 9 months ago |
| fluxxor | have common libs in root package.json and specific deps in folders | a year ago |
| freezer-js | add missed build folders on some examples | 10 months ago |
| lux | Fixed issue where code was removed in #61. Updated lux dep to ^0.8.0 | a year ago |
| marty | have common libs in root package.json and specific deps in folders | a year ago |
| material-flux | have common libs in root package.json and specific deps in folders | a year ago |

Figura 13 – Representação do Projeto Flux Comparasion

Fonte: <https://github.com/voronianski/flux-comparison>

Observa-se abaixo também a lista de bibliotecas que implementam a arquitetura flux, demonstrada pela Figura 14:

Examples

The list of Flux related implementations used in this demo.

Ready

- ☒ Facebook Flux
- ☒ Fluxible by Yahoo
- ☒ Reflux
- ☒ Alt
- ☒ Flummox
- ☒ Marty.js
- ☒ McFly
- ☒ Lux
- ☒ Material Flux
- ☒ Redux
- ☒ Redux + Flambeau
- ☒ Nuclear.js
- ☒ Fluxette
- ☒ Fluxxor
- ☒ Freezer
- ☒ Fluxury

Figura 14 – Lista de bibliotecas que implementam a arquitetura Flux

Fonte: <https://github.com/voronianski/flux-comparison>

A adoção da arquitetura Flux é divulgada não somente por desenvolvedores em pequenas aplicações. Segundo SUBRAMANYAN MURALI, o Yahoo também fez a adoção da arquitetura, em sua aplicação de cliente de e-mail (Yahoo Mail), removendo a arquitetura MVC e adotando a arquitetura Flux e a biblioteca React. O padrão MVC era o padrão de arquitetura desta aplicação, tanto no server-side como no client-side. No entanto, com qualquer base de código que tenha vários desenvolvedores mudando código ao longo de muitos anos, as coisas começam a ficar complexas.

Como qualquer arquitetura MVC, os controllers na plataforma do Yahoo Mail solicitam dados e os models principais, que por sua vez, enviam eventos para as views e os controllers. Os eventos eram a parte fundamental de funcionamento da aplicação, e foi onde foi possível detectar que o código estava ficando difícil de ser investigado (ou o melhor termo, depurado) pois a sequência de vários eventos estava gerando alterações e efeitos em cascata em toda a estrutura. Para a próxima versão da plataforma do Yahoo Mail, a equipe de engenharia tinha como objetivos: ter um Fluxo previsível dos dados, para tornar a depuração mais fácil; ter componentes implementáveis de forma independente e curva de aprendizagem mais curta para o entendimento do funcionamento da estrutura para o desenvolvimento de novas funcionalidades. O fluxo único de dados em toda a aplicação fez a equipe ter o hábito de pensar sobre todas as interações de uma aplicação, através da interface do usuário. A adição da arquitetura juntamente com a biblioteca React fez com que toda a linguagem da plataforma fosse única, o Javascript, já que no server-side a plataforma utilizava a tecnologia Node.js¹⁸. Algum tempo depois, o Yahoo implementou sua própria forma para se trabalhar com a arquitetura Flux, o Fluxible¹⁹.

Dentre as diversas aplicações e bibliotecas que possibilitaram a utilização da arquitetura Flux em projetos web, a biblioteca mais utilizada e mais recomendada é o Redux²⁰. Criado por Dan Abramov, que hoje também faz

¹⁸ Disponível em <https://nodejs.org>.

¹⁹ Disponível em <http://fluxible.io/>.

²⁰ Disponível em <http://redux.js.org/>.

parte do time de engenharia do Facebook, a biblioteca não somente permite a implementação e a utilização da arquitetura Flux em aplicações web, como conseguiu tornar alguns casos de implementação mais simples do que o proposto pela arquitetura. Para isso, a biblioteca não adere ao elemento Dispatcher, e faz uso de muitas funções puras, características do paradigma de programação funcional. O sucesso da biblioteca é reconhecido não somente pela comunidade de desenvolvedores, mas também pelos próprios criadores do Flux.

Abaixo pode-se visualizar a representação do projeto redux no Github, conforme apresenta a Figura 15:

The screenshot displays the GitHub repository for Redux. At the top, it shows the repository name 'reactjs / redux' with statistics: 1,106 watchers, 25,822 stars, and 4,413 forks. Below this, navigation tabs for Code, Issues (31), Pull requests (11), Projects (0), Pulse, and Graphs are visible. The repository description is 'Predictable state container for JavaScript apps' with a link to 'http://redux.js.org'. A summary bar indicates 2,168 commits, 9 branches, 49 releases, 382 contributors, and the MIT license. Below the summary, there are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The main section shows a list of recent commits by 'markerikson', including updates to the .github directory, build process, documentation, examples, and various source files, with timestamps ranging from 3 months ago to 6 months ago.

| Commit | Message | Time Ago |
|---------------|--|--------------|
| .github | Update ISSUE_TEMPLATE.md | 6 months ago |
| build | Update Gitbook version | 4 months ago |
| docs | Doc: Improve readability of Usage with React Router section | an hour ago |
| examples | update todos-flow example to latest versions: redux libdef (flow-type... | 28 days ago |
| flow-typed | Update argument type of reducer produced by combineReducers (#2115) | 5 days ago |
| logo | Switch out non-ASCII quotes for ASCII versions. (#1874) | 4 months ago |
| src | Add support to compose only functions in utils/compose (#2073) | 26 days ago |
| test | Add support to compose only functions in utils/compose (#2073) | 26 days ago |
| .babelrc | Close #1687, Replace es3ify with Babel ES3 transforms (#1688) | 7 months ago |
| .editorconfig | editorconfig: do not trim trailing whitespaces in Markdown files | 9 months ago |
| .eslintignore | Don't lint examples (#1928) | 3 months ago |

Figura 15 – Representação do projeto Redux no Github
Fonte: <https://github.com/reactjs/redux>

2.7 MVC X FLUX

Tendo apresentado os dois modelos de arquitetura, foi elaborado um quadro comparativo citando as características de cada um. Como são modelos

com estruturas diferentes, não é possível se criar um quadro de características comuns e fazer uma análise quantitativa sobre eles.

Quadro 1: Comparativo entre as arquiteturas MVC e Flux

| MVC | Flux |
|--|---|
| Origem: Criado na década de 1970 | Origem: Criado em 2014 |
| Cenário: Surgiu onde o uso e a necessidade das interfaces de usuário era pequena | Cenário: Surgiu onde a quantidade de aplicações client-side a cada são maiores, mais complexas, onde o usuário consegue acessar de qualquer dispositivo que possua acesso a internet e um browser |
| Pode ser utilizado em aplicações server-side (aonde possui sua maior quantidade de aplicações criadas) e client-side | Utilizado para aplicações client-side |
| Um dos padrões mais utilizados por desenvolvedores ao se criar aplicações. | Padrão ainda em fase de crescimento e de adoção por parte da comunidade de desenvolvedores. |
| Não possui rigidez em termos estruturais, permite variações (MVVM, MVP) | As implementações da arquitetura podem mudar elementos estruturais, de forma a utilizar parcialmente os conceitos os ou elementos da arquitetura. Como também não faz inferência a criação de novos elementos. Contudo, todos mantêm a proposta do Fluxo Único de Dados |
| Elementos Estruturais: Model, View, Controller | Elementos Estruturais: Action, Dispatcher, Store, View |
| Fluxo de dados: Possibilita Bidirecionalidade entre os dados da aplicação | Fluxo de dados: Um único fluxo de dados durante toda a aplicação |
| Possibilita efeitos em cascata em sua estrutura | Diminui ao máximo a possibilidade de alterações em cascata, devido ao fato do registro das ações serem sempre síncronos |
| A medida que a aplicação cresce, torna-se muito alta a complexidade do software | O entendimento do software sempre acontece mediante as <i>actions</i> que o sistema realiza. Pelo fato delas serem semânticas, e devido ao fluxo único de dados, o entendimento de toda a aplicação é mais simples |
| A medida que a aplicação cresce, torna-se difícil a manutenção | Todas as ações são únicas e não geram efeitos em cascata, o que torna mais fácil implantar features, tornando a manutenção do sistema facilitada. |
| Em aplicações client-side: Recomendado para o uso de aplicações pequenas e médias, com uma pequena | Em aplicações client-side: Bastante recomendado para o uso de aplicações com grandes possibilidades de interação, com uma estrutura maior, com uma grande quantidade de |

| | |
|---|-----------------------------|
| quantidade de estados a serem controlados | estados a serem controlados |
|---|-----------------------------|

Fonte: Elaborado pelo próprio autor.

3. METODOLOGIA

Foi realizada uma pesquisa descritiva, com o objetivo de se conhecer de forma profunda o tema do trabalho proposto. Foi feito um levantamento bibliográfico buscando conteúdos e referências sobre o tema arquitetura de softwares, e sobre as arquiteturas abordadas neste trabalho, com ênfase ao MVC e ao Flux. Por se tratar de um *modelo* de arquitetura recente, a maior quantidade de referencial sobre a arquitetura são artigos e vídeos. Também foi feito um estudo de caso com o objetivo de utilizar a arquitetura Flux de forma prática, bem como identificar suas vantagens e diferenças em prol da arquitetura MVC, mencionados em outras sessões do trabalho.

A abordagem da pesquisa tem caráter qualitativo, com o objetivo de se identificar as qualidades da arquitetura Flux e em quais cenários ou situações ela pode ser benéfica. A análise dos dados foi feita baseada no levantamento bibliográfico realizado e nos resultados obtidos no estudo de caso.

4. ESTUDO DE CASO

Nesta sessão, apresenta-se todo o estudo relatando a utilização da arquitetura Flux. O estudo está dividido em duas etapas: uma é uma citação ao próprio estudo de caso apresentado pelo Facebook em sua aplicação de chat, onde a arquitetura utilizada era o MVC, e quais os benefícios e problemas foram resolvidos ao adotarem a arquitetura Flux. Na segunda parte, será apresentado uma pequena aplicação criada pelo próprio autor, utilizando a arquitetura Flux.

Aborda-se agora o estudo de caso apresentado pelo Facebook, na conferência F8. Na implementação atual da aplicação de chat do Facebook, segundo Jing, a quantidade de ações e estados que deviam ser controlados para exibir a quantidade de mensagens que usuário havia recebido, e que ele ainda não tinha lido, bem como fornecer uma interface para que ele realizasse uma conversa com algum amigo, era muito grande.

Não existia nenhuma estrutura, era difícil adicionar novas funcionalidades pois a quantidade de código existente era grande. Todo o chat consiste no indicador de mensagens na parte superior da página, na janela de conversa individual e na página com o histórico de todas as conversas. Um dos problemas do chat era com relação ao feedback dado aos usuários quanto a quantidade de mensagens que o usuário ainda não havia visto.

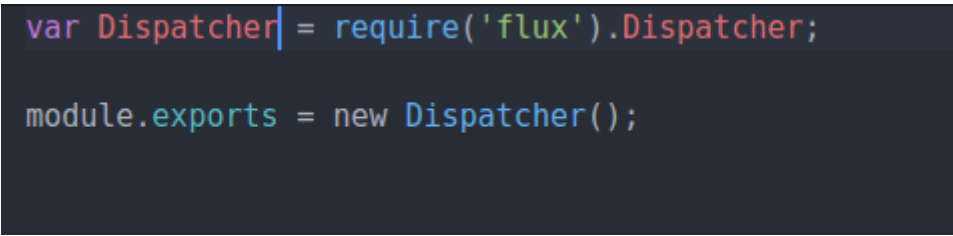
Ao clicar neste indicador, não existiam novas mensagens a serem vistas. Essa era uma das grandes reclamações dos usuários do Facebook. Sendo assim, eles redesenharam o fluxo dos dados das aplicações do chat, baseando-se nas ações realizadas pela interface, como a adição de uma nova mensagem.

Ao utilizar a arquitetura Flux, os problemas que eles possuíam com o chat foram sanados, pois a arquitetura Flux possibilita a obtenção de uma única fonte de consistência de dados, facilita a detecção de erros no sistema, já que as ações realizadas são todas conhecidas, e devido ao fato das Stores armazenarem os estados, é mais fácil realizar testes em toda a aplicação, basta sair de um estado inicial, realizar uma ação, e verificar se o estado final é de fato o estado esperado.

Nesta parte do estudo, apresenta-se uma simples aplicação, onde será possível demonstrar a utilização da arquitetura Flux. Apesar de ser citado durante o trabalho, que o Facebook indicava o uso da arquitetura para aplicações complexas e com grande quantidade de código, é inviável realizar um estudo de caso de uma aplicação ao nível de uma rede social, como é o caso do Facebook. Todo e qualquer esforço para a obtenção deste tipo de aplicação não chegaria perto da realidade.

Sendo assim, será apresenta-se uma pequena aplicação, onde é possível visualizar a utilização da arquitetura. A aplicação consiste em uma pequena lista de tarefas, onde é possível adicionar e remover elementos dessa lista. Foi escolhida este tipo de aplicação, pois é um cenário menor e mais facilmente mapeável.

Será exibido os códigos referentes as ações principais desta aplicação, bem como a identificação dos elementos desta arquitetura. Serão apresentados somente o código responsável pelo entendimento e aplicação da arquitetura Flux, conforme se demonstra pela Figura 16:



```
var Dispatcher = require('flux').Dispatcher;  
  
module.exports = new Dispatcher();
```

Figura 16: Registro do Dispatcher
Fonte: Elaborado pelo autor

O primeiro passo para a adoção e utilização da arquitetura e fazer o download do projeto através da tecnologia NodeJs. Após a instalação, é necessário adicionar o Dispatcher. Foi utilizado a biblioteca Dispatch.js, criada pelo próprio Facebook. Este elemento deverá ser o único em toda a aplicação. Abaixo visualiza-se o mapeamento das ações da aplicação pela Figura 17:

```
var keyMirror = require('keymirror');

module.exports = keyMirror({
  CREATE: null,
  REMOVE: null
});
```

Figura 17: Mapeamento das ações da aplicação
Fonte: Elaborado pelo autor

Logo após adicionar o Dispatcher, é necessário mapear as ações que a aplicação terá. Para isso, e por seguirmos os padrões de desenvolvimento recomendados pela própria comunidade de desenvolvedores, é criado um único local onde constam as ações possíveis a serem realizadas. Neste caso, as únicas ações são CREATE e REMOVE. Utilizamos a biblioteca `keymirror`²¹ apenas como um facilitador para se criar objetos do tipo chave e valor. Abaixo a exemplificação do dispatch das actions pela Figura 18:

```
var dispatcher = require('../dispatcher/dispatcher');
var constants = require('../constants/constants');

var actions = {
  create: function(text) {
    dispatcher.dispatch({
      actionType: constants.CREATE,
      text: text
    });
  },
  remove: function(id) {
    dispatcher.dispatch({
      actionType: constants.REMOVE,
      id: id
    });
  }
};
```

Figura 18: Exemplificação do dispatch das actions
Fonte: Elaborado pelo autor

Em seguida, registramos as actions, sendo identificada pelo tipo de ação que será realizada, e o seu conteúdo. A Figura 19 vem demonstrar a criação da Store:

²¹ Disponível em <https://www.npmjs.com/package/keymirror>

```

var dispatcher = require('../dispatcher/dispatcher');
var EventEmitter = require('events').EventEmitter;
var constants = require('../constants/constants');
var assign = require('object-assign');

var CHANGED = 'update';
var _items = {};

function create(content) {
  var id = (Math.floor(Math.random() * 999999)).toString(36);
  _items[id] = {
    id: id,
    text: content
  };
}

function destroy(id) {
  delete _items[id];
}

var store = assign({}, EventEmitter.prototype, {
  emitChange: function() {
    this.emit(CHANGED);
  },
  addChangeListener: function(callback) {
    this.on(CHANGED, callback);
  },
  removeChangeListener: function(callback) {
    this.removeListener(CHANGED, callback);
  }
});

```

Figura 19: Criação da Store
 Fonte: Elaborado pelo autor

Em seguida, criamos a store. Nela foi implementada a ação que deveria acontecer para cada tipo de evento. Usamos a biblioteca EventEmitter para realizar a emissão dos eventos, para que as views possam se atualizar e exibir o estado atual da Store. Assim, a Figura 20 representa a realização das ações e emissão de eventos após atualização de valores na Store:

```

dispatcher.register(function(action) {
  var content;
  switch(action.actionType) {
    case constants.CREATE:
      content = action.text.trim();
      if (content !== '') {
        create(content);
        store.emitChange();
      }
      break;
    case constants.DESTROY:
      destroy(action.id);
      store.emitChange();
      break;
    default:
  }
});

```

Figura 20: Realização das ações e emissão de eventos após atualização de valores na Store
 Fonte: Elaborado pelo autor

É feito o vínculo de cada tipo de ação, com a ação e rotinas que realmente precisam ser executadas. Ao termino da atualização dos dados, a Store faz a emissão do evento para que as views se atualizem. Abaixo, a Figura 21 representa a realização das ações e emissão de eventos após atualização de valores na Store:

```

var React = require('react');
var actions = require('../actions/actions');
var inputs = require('../inputs.react');

var main = React.createClass({
  render: function() {
    return (
      <main id="main">
        <h1>Lista de Itens</h1>
        <TodoTextInput
          id=""
          onSave={this._save}
        />
      </main>
    );
  },

  _save: function(text) {
    if (text.trim()){
      actions.create(text);
    }
  }
});

```

Figura 21: Realização das ações e emissão de eventos após atualização de valores na Store,
 Fonte: Elaborado pelo autor

Fazendo uso da biblioteca React para a criação da interface de usuário, definimos as estruturas visuais e a sua respectiva ação, nos cenários de adição de um novo elemento e a remoção de um elemento da lista. Visualiza-se assim o processo pela Figura 22 abaixo:

```
var React = require('react');
var ReactPropTypes = React.PropTypes;
var actions = require('../actions/ToDoActions');
var input = require('./ToDoTextInput.react');

var item = React.createClass({
  render: function() {
    return (
      <li key={itens.id}>
        <div>
          <button onClick={this._remove} />
        </div>
        {input}
      </li>
    );
  },
  _remove: function() {
    actions.remove(itens.id);
  }
});
```

Figura 22: Realização das ações e emissão de eventos após atualização de valores na Store
Fonte: Elaborado pelo autor

5. DISCUSSÃO E RESULTADOS

Após todo o estudo realizado sobre a arquitetura Flux, baseado em toda a bibliografia utilizada, 10 artigos demonstram a eficiência na utilização da Flux, para o desenvolvimento de aplicações cliente-side. Também foi evidenciado através do estudo de caso, que a utilização desta arquitetura permite um rápido entendimento de todo o fluxo e ações possíveis no sistema, facilitando a sua manutenção e posterior evolução.

Cabe ressaltar também que existem cenários específicos para se obter os benefícios da arquitetura em uma aplicação. Caso a sua aplicação trabalhe com dados estáticos, ou que raramente mudam, ou que não existe a necessidade de utilização de cache dos dados da aplicação, e que a aplicação não seja altamente complexa com uma grande quantidade de código, a arquitetura Flux não trará benefícios para a sua aplicação. Nestes cenários, o padrão MVC irá atendê-lo perfeitamente. Contudo, mesmo em aplicações não tão grandes, como um caso de uma rede social, é possível utilizar a arquitetura, baseado nos cenários reportados por Dan Abramov. Segundo DAN ABRAMOV, a arquitetura Flux poderá ser bastante útil em sua aplicação, caso os seus dados sofram mudança ao longo do tempo, e se sua aplicação necessita de exibi-los em real integra ao valor contido na sua estrutura de dados; também recomenda a adoção em casos onde os dados são relacionais e os Models dependem um do outro para existir, se os mesmos dados são refletidos em diversos locais da interface, de formas diferentes.

Percebe-se que o Flux é uma inovação frente a um modelo altamente consolidado e amplamente utilizado em termos de arquitetura de aplicações. Aplicações baseadas em modelos de eventos tem sido bastante utilizada, seguindo os padrões CQRS e EventSourcing, segundo MICHAEL RIDLAND. O Flux é sem dúvidas, uma possível implementação deste modelo para aplicações cliente-side. Mesmo assim, a sua adoção ou surgimento não inferem na não utilização do modelo MVC, que pode e deverá ser usado em muitos cenários no desenvolvimento de aplicações. A evolução do Flux, sua maior adoção e se de fato esta arquitetura é mais eficiente que o MVC, em se tratando de aplicações cliente-side, só poderá ser observada após muitos estudos e aplicações.

6. CONSIDERAÇÕES FINAIS

A importância de uma arquitetura para o desenvolvimento de aplicações sempre será necessário, pois ela possibilita a evolução de uma aplicação. Foi apresentado o já consolidado modelo MVC, reconhecendo a sua importância para o desenvolvimento das aplicações, como também foi apresentado a nova proposta de arquitetura para aplicações client-side, o Flux, que possui como grande vantagem o fluxo único de dados.

REFERÊNCIAS

ADDY OSMANI, 2015. Disponível em: <https://addyosmani.com/resources/essentialjsdesignpatterns/book/>. Acesso em 20 set. 2016.

ALMEIDA, Rodrigo Rebouças de. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/mvc/mvc.htm>. Acesso em 20 set. 2016.

AMIR SALIHEFENDIC. Disponível em: <https://medium.com/hacking-and-gonzo/flux-vs-mvc-design-patterns-57b28c0f71b7#.9ybv71fcq>. Acesso em 20 set. 2016.

BAPTISTELLA, Jose Adriano. Disponível em: <http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx>. Acesso em 20 set. 2016.

BORINI, Stefano. *Understanding Model View Controller*. Disponível em: <https://www.gitbook.com/book/stefanoborini/modelviewcontroller/details>. Acesso em 20 set. 2016.

BRAGGE, Matti. ***Model-View-Controller architectural pattern and its evolution in graphical user interface frameworks***. Bachelor's Thesis, 2013.

DAN ABRAMOV. Disponível em: <https://medium.com/swlh/the-case-for-flux-379b7d1982c6#.yzypdtsjv>. Acesso em 20 set. 2016.

FOWLER UI. Disponível em: <http://www.martinfowler.com/eaDev/uiArchs.html>. Acesso em 20 set. 2016.

FOWLER, Martin; RICE, David; FOEMMEL, Matthew; HEATT, Edward; MEE, Robert, STAFFORD, Randy. **Patterns of Enterprise Application Architecture**. Addison-Wesley Professional, 2003.

<http://pensandonaweb.com.br/como-funciona-a-internet-e-a-world-wide-web/>
Acesso em 10/10/2016

<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/mvc/mvc.htm>.
Acesso em: 22/10/2016

<https://facebook.github.io/flux/docs/overview.html> – **Flux Documentation**.
Acesso em: 22/08/2016

<https://facebook.github.io/react/> - **React Documentation**, Acesso em: 20/10/2016

<https://www.youtube.com/watch?list=PLb0IAmt7-GS188xDYE-1ShQmFFGbrk0v&v=nYkdrAPrdcw> - **Hacker Way: Rethinking Web App Development at Facebook**. Acesso em: 22/09/2016

MARK RICHARDS. **Software Architecture Patterns**; O'Reilly, 2015.

MICHAEL RIDLAND. Disponível em: <http://www.michaelridland.com/xamarin/mvvm-mvc-is-dead-is-unidirectional-a-mvvm-mvc-killer/> / Acesso em 20 set. 2016.

PAUL COWAN. Disponível em: <http://www.thesoftwaresimpleton.com/blog/2013/03/23/client-side-mvc/> Acesso em 20 set. 2016.

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Pearson Education do Brasil, 1995.

RICHARDS, Mark. **Software Architecture Paterns: Understanding Architecture Patterns and When to use Them**. Sebastopol: O'reilly Media, 2015.

SANTOS, Marcel. Disponível em: <http://tableless.com.br/como-funciona-internet-e-world-wide-web/>. Acesso em 20 set. 2016.

SOMMERVILLE, Ian. **Engenharia de Software**. São Paulo: Person Education, 2003.

SUBRAMANYAN MURALI. Disponível em: <https://yahooeng.tumblr.com/post/101682875656/evolving-yahoo-mail>. Acesso em 20 set. 2016.

TECHTUDO, 2006. Disponível em: <http://www.techtudo.com.br/artigos/noticia/2013/04/internet-completa-44-anos-relembre-historia-da-web.html>. Acesso em 20 set. 2016.

TRYGVE, Reenskaug, 1979. Disponível em: <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>. Acesso em 20 set. 2016.

TRYGVE, Reenskaug. Disponível em: <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>. Acesso em 20 set. 2016.

WIKIPEDIA, 2006. Disponível em: https://pt.wikipedia.org/wiki/Hist%C3%B3ria_da_Internet. Acesso em 20 set. 2016.