UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ADOLFO HENRIQUE SCHNEIDER

Desenvolvimento web com Client Side Rendering: combinando Single Page Application e serviços de backend

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Marcelo Soares Pimenta

Porto Alegre
2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Carlos Arthur Lang Lisbôa
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Primeiramente agradecer o meu orientador, Marcelo Soares Pimenta, por dedicar seu tempo me mostrando como a abordagem desse trabalho poderia ser estruturada.

À minha família por ter sempre me apoiado e compreender as dificuldades atreladas à dedicação em um trabalho como este.

À minha mãe, Shirley Schneider, por ter sempre acompanhado de perto a minha educação e me encorajado a ser curioso e experimentalista.

Aos amigos do ISI da empresa ADP, com quem adiquiri muito do conhecimento exposto neste trabalho.

Ao grande amigo Thom Morais por ter me ajudado com a confecção das imagens.

E, especialmente, à Taís Bellini, por todas as revisões, por sugerir alterações quando necessárias e por ter mostrado, diariamente, como conectar os pontos e ligar as ideias. Mas, além disso, por estar sempre por perto demonstrando o seu amor e dando todo o suporte emocional.

RESUMO

Esse trabalho consiste em uma análise do uso de *Client Side Rendering* para desenvolvimento web, através das Single Page Applications, e seus benefícios em relação a abordagem de *Server Side Rendering*. Para isso, fizemos experimentos comparando o uso de *Server Side Rendering* puro, esse mesmo com o auxílio de AJAX e uma aplicação *Client Side* utilizando somente AJAX. Além disso, mostramos o forte papel da Composição de Objetos do Javascript em permitir aplicações leves que possam rodar no cliente. Por outro lado, mostramos, através da análise do Firebase, que o uso de serviços de backend nesse tipo de arquitetura devem ser acompanhados de precauções em termos de segurança, uma vez que o cliente tem acesso direto a serviços como banco de dados.

Palavras-chave: Server Side Rendering, Client Side Rendering, Single Page Application, Backend as a Service, AJAX, Firebase, Javascript.

ABSTRACT

This document is an analysis of the *Client Side Rendering* approach for web development, through the use of Single Page Applications, and its benefits as opposed to *Server Side Rendering*. For this purpose, we developed experiments comparing an application using pure *Server Side Rendering* to a version with the support of AJAX and another entirely rendered in the *Client Side*. Besides, we present how the Javascript's Object Composition role is important in this scenario to allow the development of lightweight applications that can run in the client-side. On the other hand, we demonstrate, through a Firebase analyisis, that the use of backend services in this type of architecture must be followed by security precautions, once the client has direct access to services such as the database.

Keywords: Server Side Rendering, Client Side Rendering, Single Page Application, Backend as a Service, AJAX, Firebase, Javascript.

LISTA DE FIGURAS

Figura 2.1 Anatomia da Assincronicidade no Javascript 13
Figura 2.2 Interação Assíncrona 15

Figura 3.1 Marcações Server Side Rendering 17
Figura 3.2 Arquitetura Server-Side Rendering 18
Figura 3.3 Interação Server Side Rendering 19

Figura 4.1 Marcações Client Server 21
Figura 4.2 Arquitetura Client Side Rendering 22

Figura 5.1 Hulk Comic Viewer 31
Figura 5.2 Linha de tempo 33
Figura 5.3 Frames do experimento server side 33
Figura 5.4 Linha do tempo server side com ajax 34
Figura 5.5 Frames do experimento server side com ajax 35
Figura 5.6 Linha do tempo client side 35
Figura 5.7 Frames do experimento client side 36
Figura 5.8 Código da aplicação minificado 37
Figura 5.9 Procurando servidores acessados pela aplicação 37
Figura 5.10 Dados da aplicação acessíveis 38
Figura 5.11 Dados da aplicação alteráveis 38
Figura 5.12 Firebase - Acesso Negado 39

LIST OF TABLES

Tabela 4.1 Objetos e suas ações..30

LISTA DE ABREVIATURAS E SIGLAS

SPA	Single Page Application
BaaS	Backend as a Service
AJAX	Asynchronous Javascript And XML
MVC	Model View Controller
JSON	JavaScript Object Notation
JWT	JSON Web Token
API	Application Programming Interface
SDK	Software Development Kit
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
RTT	Round Trip Time

CONTENTS

1 INTRODUÇÃO **10**
1.1 Objetivos **11**
1.2 Estrutura **11**
2 CONCEITOS BÁSICOS **12**
2.1 Arquitetura cliente-servidor **12**
2.2 Javascript e Ajax **12**
2.2.1 Assicronicidade do Javascript 12
2.2.2 Ajax 14
2.3 Bancos de dados orientados a documentos **14**
2.3.1 Durabilidade 15
2.3.2 Consistência 16
3 SERVER SIDE RENDERING **17**
3.1 Problemas **18**
3.1.1 Escalabilidade 18
3.1.2 Experiência do Usuário 19
4 CLIENT SIDE RENDERING **21**
4.1 Backend as a service para SPA **23**
4.1.1 Alta performance 24
4.1.2 Segurança 25
4.2 A flexibilidade do Javascript **27**
5 EXPERIMENTOS **31**
5.1 Experimento 1 : Hulk Comic Viewer **31**
5.1.1 Server Side Rendering sem Ajax 32
5.1.2 Server Side Rendering com Ajax 34
5.1.3 Client Side Rendering 35
5.2 Experimento 2: Segurança client side **36**
5.3 Experimento 3: Complexidade das Security Rules **39**
5.4 Experimento 4: Performance banco de dados em tempo real **41**
6 CONCLUSÃO **43**
BIBLIOGRAPHY **45**
APPENDIX A — REGRAS DE SEGURANÇA DO BABIES BOOK **47**

1 INTRODUÇÃO

O mercado de arquiteturas de aplicações web passa por um momento de grande euforia. Novos frameworks e padrões de desenvolvimento são apresentados constantemente para uma comunidade de desenvolvedores que, por muitas vezes, acaba se sentindo perdida devido as incertezas e imaturidades atreladas a cada uma dessas opções. Dentre as muitas opções emergentes, já é possível traçar um padrão entre as soluções mais buscadas. Tanto os modelos de Single Page Application (SPA), que consistem em aplicações que carregam dinamicamente o conteúdo sem necessidade de atualização da página, quanto serviços de cloud para backend — Backend as a Service (BaaS) —, que provêem armazenamento de dados e outras funcionalidades para sistemas, tem se mostrado uma tendência nas aplicações mais atuais.

Essa tendência revela uma mudança radical em relação ao paradigma clássico de aplicações web que usam a arquitetura cliente-servidor, na qual o cliente é o navegador do usuário. O modelo tradicional de *Server Side Rendering*, onde o servidor é responsável por processar e renderizar a maior parte do conteúdo, foi amplamente utilizado por muitos anos. Porém, com o crescimento da abrangência da internet e do número de usuários acessando as páginas, esse modelo acabou se mostrando ineficiente. Como há um componente centralizado que executa a maior parte das ações de uma aplicação, um grande número de requisições sobrecarrega o servidor, o que afeta a experiência do usuário por sua falta de performance. A partir dessa necessidade, as aplicações de *Client Side Rendering*, representadas pelas SPAs, que executam e renderizam no lado do cliente, em combinação com soluções de BaaS, começaram a ganhar espaço e dominar o mercado de desenvolvimento web, por serem mais performáticas e flexíveis quando comparadas ao modelo centralizado no servidor.

Uma vez que as SPAs trazem o processamento dos dados para o lado do cliente, a aplicação se torna mais descentralizada, melhorando a performance e, consequentemente, a experiencia do usuário. Porém, o lado do cliente é mais vulnerável, pois se apresenta aos usuários de forma menos protegida e é possível acessar o código que está sendo executado. Isso traz a responsabilidade de tomar precauções que asseguram que o navegador terá acesso somente ao conteúdo a ele destinado. É nesse cenário que os BaaS se mostram grandes aliados às SPAs

pois, como veremos no decorrer desse trabalho, além de entregarem alta performance e escalabilidade, possuem ferramentas que diminuem as vulnerabilidades de segurança.

1.1 Objetivos

Esse documento tem como principal objetivo apontar os benefícios do modelo de *Client Side Rendering* em relação ao de *Server Side Rendering*. Além disso, entender a relação do Javascript na criação de aplicações que podem ser processadas no cliente. E ainda, mostrar que, para aproveitar o potencial das SPAs, é necessário que os serviços de backend acompanhem essa mudança, provendo serviços de alta performance e ferramentas de segurança uma vez que o cliente vai acessá-las diretamente.

1.2 Estrutura

No capítulo 2 veremos os conceitos básicos necessários para entender a dissertação. Em seguida, no capítulo 3, apontaremos dois dos principais problemas do modelo de *Server Side Rendering* - escalabilidade e experiência do usuário. Já no capítulo 4, entenderemos como as soluções de *Client Side Rendering* resolvem os problemas previamente identificados. Mostraremos o papel do Javascript na resolução dos problemas e que, por ventura da natureza das SPAs são necessários serviços de Backend voltados para esse modelo. Assim, ainda no capítulo 4 iremos entender, analisando o *Firebase*[1], como esses BaaS satisfazem tais necessidades. E por último, com o inuito de demonstrar e validar os pontos levantados no decorrer do documento, o capítulo 5 traz experimentos voltados para os tópicos abordados.

[1] https://firebase.google.com/

2 CONCEITOS BÁSICOS

Neste capítulo serão apresentados alguns conceitos considerados fundamentais para a compreensão do trabalho, incluindo Arquitetura cliente-servidor (seção 2.1), JavaScript e Ajax (seção 2.2) e Bancos de dados orientados a documentos (seção 2.3).

2.1 Arquitetura cliente-servidor

Em redes de computadores, o modelo cliente-servidor (MITCHELL, 2016) consiste em diferentes dispositivos efetuando chamadas entre eles através de protocolos como o *HTTP*. O cliente é o dispositivo que efetua a chamada e o servidor é aquele que recebe e, possivelmente, responde. Portanto, é possível que um mesmo dispositivo possa se comportar tanto como um servidor quanto como um cliente. Em aplicações web, o cliente é um dispositivo que, através de um navegador, requisita conteúdos a um endereço (*URL*) que referencia um servidor. Esse é responsavel por responder ao cliente com conteúdo que é interpretado pelo navegador e mostrado ao usuário final.

2.2 Javascript e Ajax

Por muito tempo o Javascript foi tido como uma linguagem auxiliar de manipulação de elementos de tela para aplicações web. Tal cenário foi se alterando conforme a evolução da linguagem e consequente evolução dos navegadores, uma vez que esses executam interpretadores Javascript. Nessa seção, vamos entender como essa evolução impactou de forma considerável a forma como as páginas interagem com o usuário.

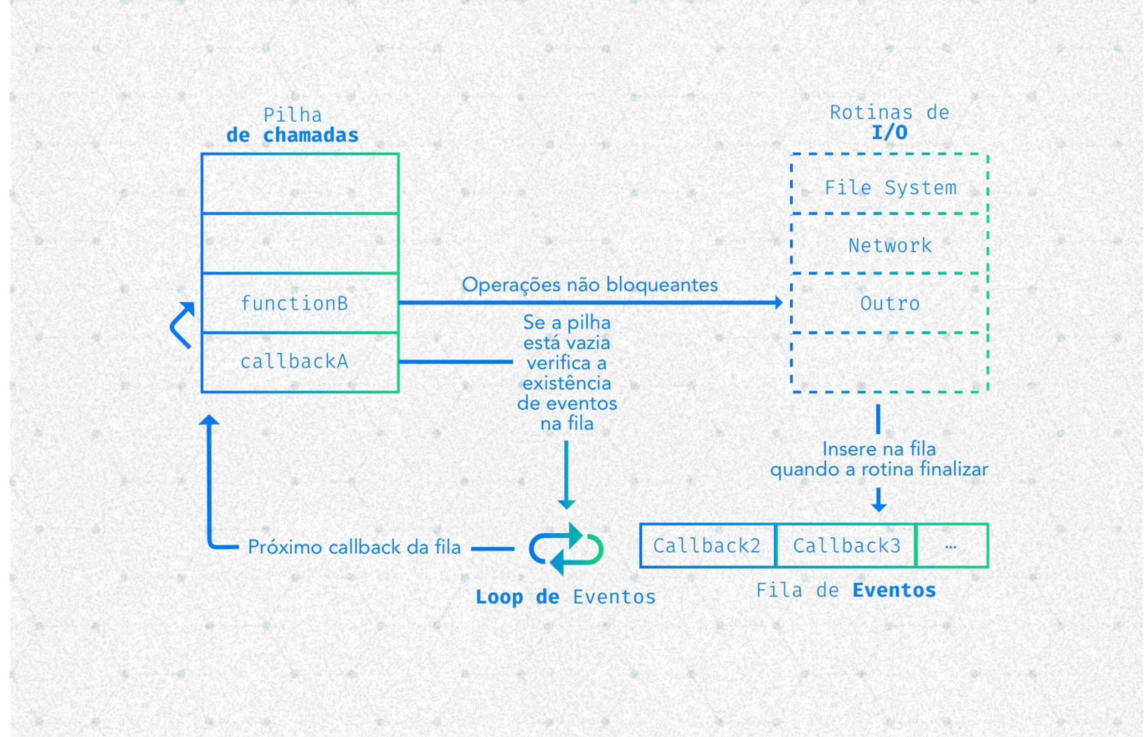
2.2.1 Assicronicidade do Javascript

A característica mais marcante do Javascript é sua natureza assíncrona. O que pode ser considerado surpreendente para muitos é que, apesar de o Javascript possibilitar aplicações com interações não bloqueantes ao usuário, ele é execu-

tado em uma *thread* única. Portanto, é falso afirmar que a linguagem proporciona assincronia através do uso de múltiplas *threads*. Aplicações que usam Javascript têm seu código executado com base em uma **pilha de chamadas**, onde a *thread* empilha todas as chamadas aninhadas e desempilha conforme as executa.

Assim como em qualquer outra linguagem de programação, um programa em Javascript pode possuir chamadas de I/O — entradas do usuário ou respostas de requisições HTTP são exemplos comuns. A diferença no caso do Javascript é que, nessas situações, ele não espera o retorno da chamada e apenas segue empilhando rotinas e as executando até que não as encontre mais. Para tratar o retorno de chamadas de I/O, são usados os *callbacks* — subrotinas que são definidas juntamente com a chamada. Toda vez que uma função de I/O retorna, o seu *callback* entra para a **fila de eventos** e é empilhado na pilha de chamadas assim que essa esvaziar (MARTENSEN, 2015).

Figure 2.1: Anatomia da Assincronicidade no Javascript



Na figura 2.1, temos um exemplo de como a pilha de chamadas e a fila de eventos se comunicam. No caso ilustrado, como a pilha de chamadas estava anteriormente vazia e a fila de eventos possuia *callbacks* a serem executados, o próximo item da fila entra para a pilha de chamadas. Dessa forma, o *callback* é executado e, caso hajam requisições a novas chamadas assíncronas, essas entrarão

para a fila de eventos assim que completadas. Enquanto o programa Javascript estiver executando, esse processo ocorre indefinidas vezes no que chamamos de **loop de eventos**.

Considerando que o Javascript roda em uma única *thread*, é essencial que o desenvolvedor dobre os cuidados ao estruturar seu código que rodará no cliente, dado o fato de que rotinas de alta complexidade poderão travar a página durante sua execução. Com o objetivo de tratar esse problema, alguns navegadores já provêem uma API interna para rodar Javascript em *threads* múltiplas, os Web Workers (GRAVELLE, 2016).

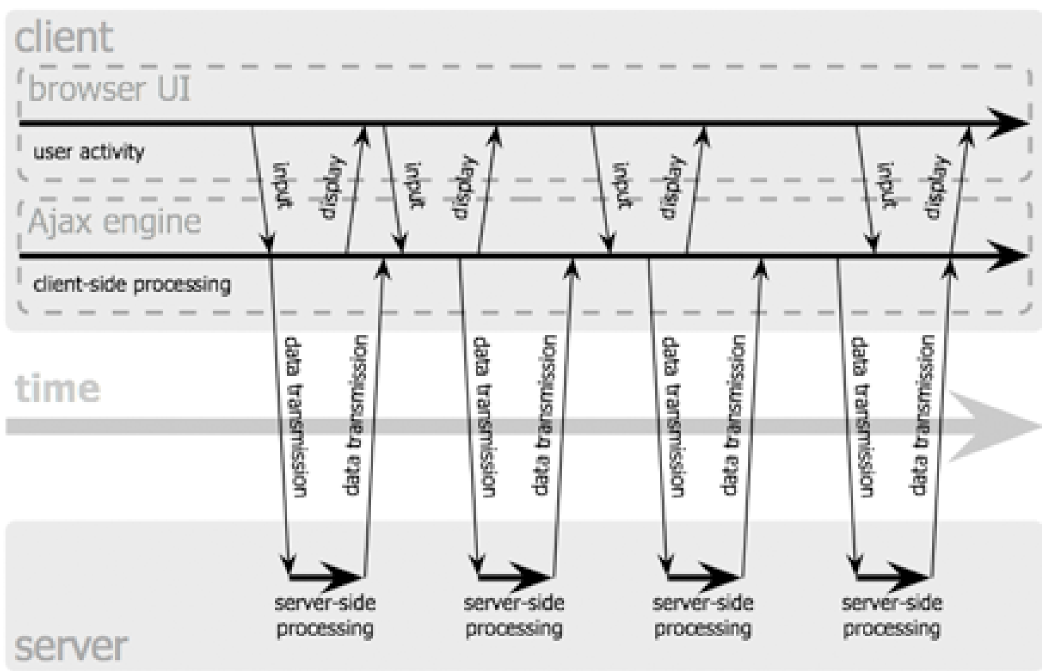
2.2.2 Ajax

Asynchronous Javascript And XML (Ajax) (GARRETT, 2005), faz uso das características assíncronas do Javascript para dar mais dinâmica às interações entre cliente e servidor. A principal ferramenta que possibilitou o Ajax foi o objeto *XMLHttpRequest*, desenvolvido pela Microsoft, que permitiu que o navegador fizesse chamadas *HTTP* para um servidor sem a necessidade do recarregamento da página. Através dessa prática, o cliente pode efetuar uma requisição ao servidor de aplicação que expõe um serviço, receber a respectiva resposta e processá-la no navegador do usuário, fazendo alterações no HTML ou CSS. Como vimos, tal operação é não bloqueante ao usuário. Portanto, enquanto o navegador aguarda a resposta do servidor, pode executar outras operações e inclusive fazer novos requisições. Desse modo, assim que o servidor responde, uma rotina de *callback* entra na fila de eventos do Javascript e ela será responsável por fazer as alterações na tela com base no conteúdo da resposta. A figura 2.2 ilustra uma interação assíncrona entre usuário e aplicação.

2.3 Bancos de dados orientados a documentos

As soluções de bancos de dados orientados a documentos trazem, em determinados aspectos, algumas vantagens em relação ao modelo tradicional de banco de dados relacional. O principal deles é a flexibilidade quanto a estruturação do banco. Ao contrário da abordagem relacional, que é definida através de

Figure 2.2: Interação Assíncrona



um esquema, nesse modelo não temos um esquema, permitindo que os dados salvos tenham diferentes estruturas e que essas estruturas possam facilmente ser alteradas durante o contínuo desenvolvimento da aplicação.

Outro benefício muitas vezes apontado nesse modelo é sua capacidade de escalar. Ao contrário do modelo relacional, os bancos orientados a documentos não possuem grande quantidade de mecanismos que asseguram **consistência** e **durabilidade**. Esse fato, portanto, possibilita uma melhor capacidade de escalar, porém expõe a aplicação a outros diferentes problemas que veremos a seguir.

2.3.1 Durabilidade

Quando falamos de durabilidade nesse contexto, estamos querendo garantir que o dado irá permanecer inalterado até que uma nova operação o manipule. Em bancos de dados orientados a documentos, perdemos a garantia de durabilidade visto que as operações de escrita não são imediatamente gravadas em disco. Portanto, como estas podem permanecer por certo tempo salvos somente em memória, uma queda de luz podem causar em perda de dados.

2.3.2 Consistência

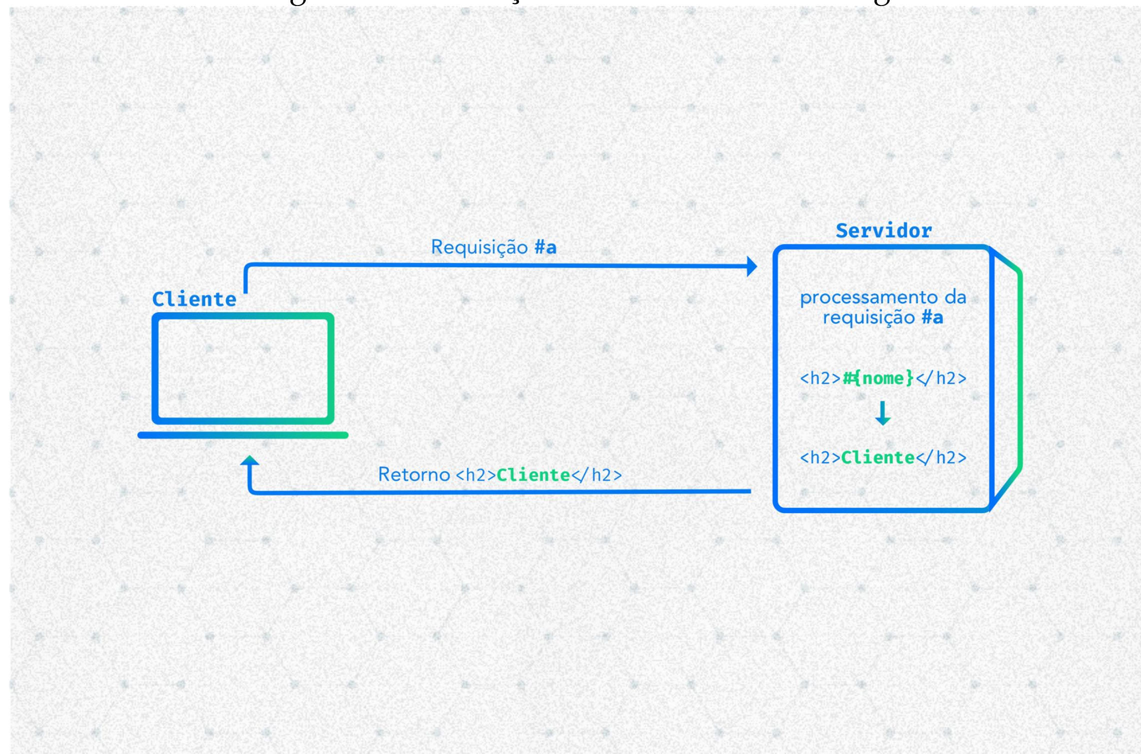
A consistência de um banco diz respeito a sua propriedade de, antes de uma transasão ocorrer, estar com os dados consistentes com as transações anteriores. Manter essa característica se torna uma tarefa complexa quando temos o banco dividido em partições devido a quantidade de dados armazenados, pois precisamos ordenar as transações que ocorrem entre diferentes partições. Bancos baseados em documentos tendem a relaxar esse fator de forma que a consistência não esteja sempre garantida, chamamos essa abordagem de consistencia eventual visto que, eventualmente, as alterações estarão distribuidas entre todas as partições.

Um exemplo de aplicação que faz uso de consistência eventual é o Twitter. Dado que para cada novo *tweet* é necessário atualiazar a *timeline* de cada um dos seguidores, não precisamos garantir que todas sejam alteradas simultaneamente, podendo acontecer de outros *tweets* serem escritos de maneira não serializada. Vale ressaltar que o relaxamento da consistência pode não ser desejável para outros tipos de aplicação. Em sistemas bancários, por exemplo, precisamos garantir que alterações em saldos ocorram de forma serializada.

3 SERVER SIDE RENDERING

Em aplicações para a web, o conceito de *Server Side Rendering* se baseia em uma arquitetura que possui um cliente - navegador, que efetua requisições; e um servidor, o qual é responsável por analisar e processar essas requisições. A partir disso, o servidor responde com um arquivo HTML que será interpretado pelo navegador e apresentado ao usuário final. O desenvolvimento de aplicações nesse modelo passa por construir arquivos HTML que possuem marcações que funcionam como *placeholders* para dados dinâmicos. Tais marcações são traduzidas, em tempo de execução do servidor, para dados voltados para o cliente que disparou a requisição - ver figura 3.1. Portanto, em modelos de *Server Side Rendering* puros, o cliente é simplesmente o agente que faz requisições e recebe respostas.

Figure 3.1: Marcações Server Side Rendering



Uma vez que a lógica da aplicação está contida no servidor, para cada nova requisição é criada uma *thread* que a processará. Além disso, é comum gerar um identificador(ID) de sessão para cada cliente. Esse ID é salvo em um *cookie* que, para fins de identificação, é enviado ao servidor a cada nova requisição. A sessão, por sua vez, é um objeto salvo em memória no servidor que contém toda

a informação do cliente que seja relevante à aplicação. Como ilustrado na figura 3.2, as *threads* também são responsáveis por buscar dados em serviços externos ao servidor de aplicação quando necessário.

Figure 3.2: Arquitetura Server-Side Rendering

3.1 Problemas

O constante uso desse modelo mostrou seus pontos fracos, os quais viraram motivação para diferentes propostas de modelagem de aplicação. Nesse documento, iremos expor dois desses pontos e nos capítulos seguintes mostraremos como a abordagem de *Client Side Rendering* os resolvem.

3.1.1 Escalabilidade

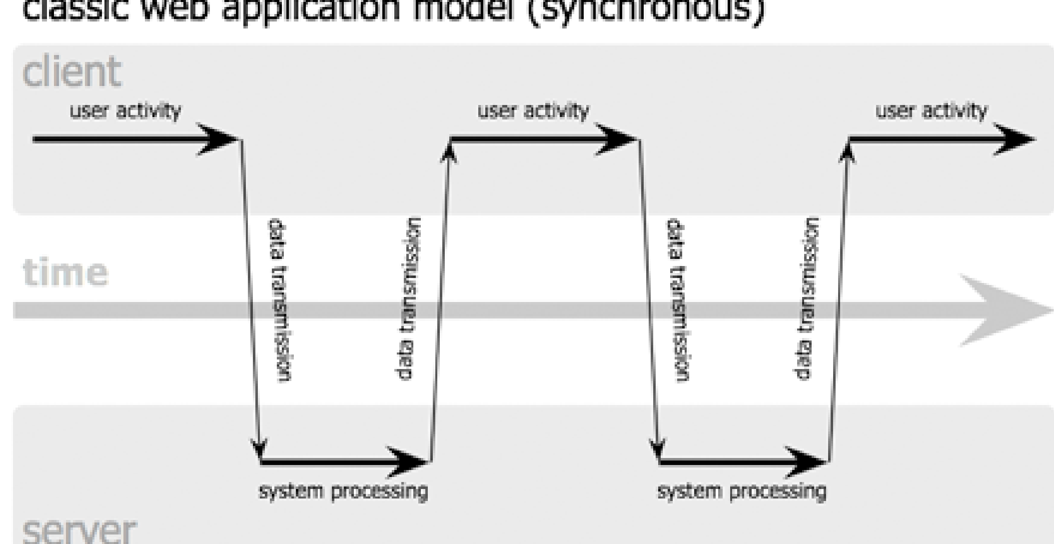
Escalar uma aplicação centrada no servidor é suscetível a problemas de performance, pois é ele que está fazendo todo o processamento das requisições feitas para a aplicação. No momento em que muitos usuários estão efetuando chamadas ao servidor, o número de *threads*, bem como a quantidade de sessões

armazenadas, podem chegar no limite suportado pelo servidor e os usuários começarão a encontrar problemas ao interagir com a aplicação. Nesse caso, torna-se indispensável o constante aumento do poder de computação através da configuração de servidores adicionais, juntamente com mecanismos de *load balancing* (JR., 2012) que distribuem a carga de trabalho entre os servidores disponíveis. Para tornar esse processo menos custoso e mais flexível às necessidades de cada aplicação, serviços de *cloud* possibilitam que servidores sejam iniciados e terminados conforme a demanda da aplicação (AMAZON, 2016). Entretanto, no capítulo 4, veremos que utilizar *Client Side Rendering* é uma alternativa menos custosa pois dispensa a necessidade de um servidor rodando a lógica de aplicação da página.

3.1.2 Experiência do Usuário

Nesse tipo de aplicação, as interações do usuário que envolvem processamento de dados contido em algum servidor implicam em uma completa re-renderização da página. Esse fato pode tornar a experiência do usuário desagradável, uma vez que o usuário terá que esperar o intervalo de sua requisição chegar ao servidor da aplicação para toda ação que fizer, somado ao tempo de processamento da requisição e do recebimento da resposta. A figura 3.3 ilustra essa interação.

Figure 3.3: Interação Server Side Rendering

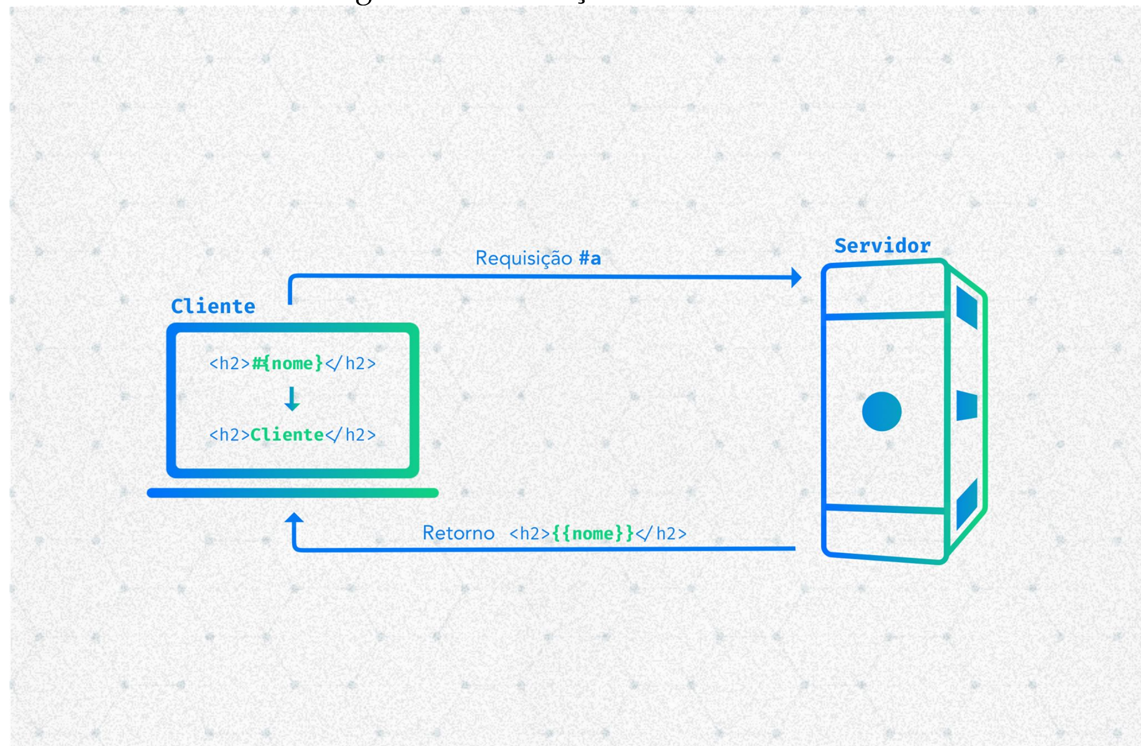


Dada essa natureza síncrona da abordagem de *Server Side Rendering*, a especificação Ajax, faz uso das características assíncronas do Javascript para dar mais dinâmica às interações entre cliente e servidor. Assim, a adoção do Ajax se iniciou combinando com aplicações que rodavam no servidor. Portanto, é importante notar que, apesar do aumento da participação do cliente no processamento da aplicação, ela ainda está majoritariamente contida no servidor. É nele que está armazenado o estado da aplicação e somente ele poderá dizer o que deve acontecer para cada ação do usuário. Uma troca de página que demande alteração da *URL*, por exemplo, obrigatoriamente disparará uma nova requisição para renderização e carregamento da página alvo. Por outro lado, conseguimos desacoplar significativamente a aplicação do servidor, proporcionando maior flexibilidade e melhorando consideravelmente a experiência do usuário. Por esse motivo, podemos afirmar que a arquitetura de *Server Side Redering* com Ajax é o limiar para o modelo de *Client Side Rendering*, que entenderemos melhor no próximo capítulo.

4 CLIENT SIDE RENDERING

Com a evolução proporcionada pelo Ajax, desenvolvedores começaram a experimentar a criação de aplicações inteiramente contruidas com essa tecnologia. Dessa forma, surgiu o conceito de *Client Side Rendering*, na qual o servidor não é mais responsável pela lógica da página e temos aplicações sendo renderizadas inteiramente no cliente. Uma aplicação baseada puramente em *Client Side Rendering* delega ao cliente o trabalho de processar todas as dependências de dados contidas em uma página. Assim, o cliente inicialmente recebe uma página HTML contendo marcações de dados que ele próprio terá de resolver, como pode ser observado na figura 4.1, diferente das aplicações web de *Server Side Rendering*, onde o cliente carrega o HTML já processado. Em nosso estudo vamos nos conter às aplicações voltadas para navegadores que rodam em Javascript, que são comumente chamadas de *Single Page Applications* (SPAs), pois não recarregam a página durante a interação com o usuário.

Figure 4.1: Marcações Client Server

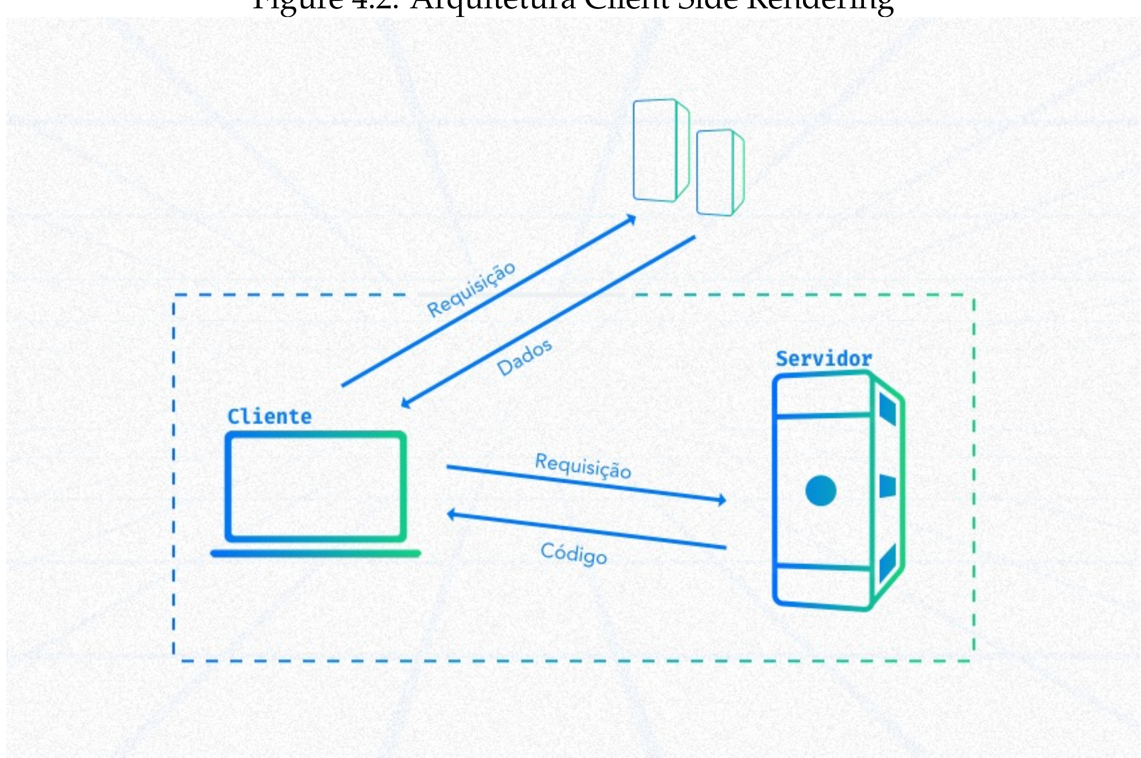


Na figura 4.1, podemos observar a diferença de abordagem na resoluções dos dados entre as duas arquiteturas. No *Client Side*, estamos fazendo a requisição do arquivo HTML e recebendo ele ainda com as dependências, as quais

são resolvidas no navegador. Ao contrário disso, em uma abordagem *Server Side* o cliente só possui acesso ao HTML depois que este tenha os dados resolvidos ainda no servidor.

No inicio do desenvolvimento das SPAs, que teve seus primeiros exemplares em meados dos anos 2000 (LANGE, 2016), não existia consenso quanto à arquitetura de uma aplicação desse gênero. Assim, cada desenvolvedor empregava os seus métodos e, consequentemente, a experiência de desenvolvimento não refletia a experiência entregue ao usuário. Dar manutenção a uma SPA demandava muito esforço. Foi assim que surgiu a necessidade da criação de *frameworks* baseados no modelo Model-View-Controller (MVC) que estabeleceram métodos estruturados de como desenvolver esse tipo de aplicação. Esses *frameworks*, tais como Backbone, Angular, Ember e React, popularizaram e facilitaram o uso da arquitetura.

Figure 4.2: Arquitetura Client Side Rendering



Em contraponto ao que vimos na figura 3.2, a figura 4.2 mostra que o servidor contendo o código da aplicação deixa de ser o centro da arquitetura, delegando ao cliente o papel de processar a lógica e requisitar serviços de outros servidores para anexação de informações dinâmicas. Portanto, as *threads* de sessão do modelo clássico de *Server Side Rendering* não se fazem mais necessárias

uma vez que cada cliente utiliza do seu poder de computação para executar a aplicação. Nesse contexto, podemos afirmar que aplicações que usam arquiteturas baseadas em *Client Side Rendering* desafogam a carga de trabalho do servidor, pois esse deixa de processar a lógica de programação da página e torna-se exclusivamente um fornecedor de conteúdo estático.

Além de diminuir a carga de trabalho do servidor, uma aplicação que utiliza Ajax para todas as chamadas ao servidor proporciona interações assíncronas inclusive na troca de páginas. Assim, desfrutamos de uma experiência onde trocamos de páginas sem esperar pela resposta do servidor e podemos inclusive manter o estado dessas páginas. Isso significa que é possível efetuar uma requisição em uma página, acessar outra e, quando voltarmos para a página original teremos a resposta da requisição anterior caso esta tenha terminado. Na seção 5.1.3, veremos um experimento mostrando essa situação na prática.

Uma vez que teremos diferentes servidores que serão responsáveis por prover serviço de banco de dados e outras questões sensíveis a aplicação, queremos que estes sejam robustos e, além disso, acompanhem a fluidez da experiência que a SPA proporciona. É importante ressaltar também que esses serviços devem contar com uma eficiente camada de segurança, visto que o código da aplicação está aberto ao cliente através do navegador. Com isso, veremos nesse capítulo uma proposta de solução para *backend* no contexto dessa arquitetura e, em seguida, vamos mostrar como a flexibilidade do Javascript tem importante papel em prover os *frameworks* de SPA.

4.1 Backend as a service para SPA

Uma vez que em *Single Page Applications* o servidor apenas provê páginas estáticas para o navegador, são necessários servidores terceiros que forneçam serviços como banco de dados ou envio de emails. Dessa forma, dado o crescimento da adoção do padrão SPA, grandes empresas do mercado de tecnologia tem investido em fornecer tais serviço para essas aplicações. Essas empresas, como *Google*, *Microsoft* e *Amazon* fornecem *Backend as a Service* (BaaS) focados em SPA de forma que entreguem alta performance e segurança.

Nesse estudo nos conteremos a analisar o *Firebase*, um serviço de banco de dados em tempo real fornecido pelo *Google*. Em seguida, portanto, analisaremos

como o *Firebase* provê performance e segurança.

4.1.1 Alta performance

Uma vez que as SPAs fornecem interações sem recarregamento de página, precisamos de serviços de banco de dados que acompanhem a fluidez da interação com a aplicação. Portanto, a performance é uma questão chave nesse cenário. Para alcançar isso, o *Firebase* optou por fornecer uma estrutura de banco de dados orientado a documentos onde temos dados salvos em estruturas JSON. Tal opção é produto de sua flexibilidade, pois, como vimos na seção 2.3, podemos ter *schemas* diferentes para cada entrada no banco visto que um documento não é estruturado. Isto é, ao contrário do modelo de banco de dados relacional, onde cada entrada do banco possui as mesmas colunas, esse modelo possibilita que as entradas tenham diferentes estruturas com diferentes tipos de dados.

Através dessa estrutura, o *Firebase* possibilita que o desenvolvedor faça escrita e leituras em nodos específicos do documento. Portanto, em uma leitura não é necessário carregar todos os subnodos de um objeto se estivermos interessados em ler o conteúdo de uma única chave.

Código 1 Exemplo de estrutura de banco no Firebase

```
{
    "post" : {
        "uid" : {
            "postId1" : {
                "title" : "Title1",
                "date" : "20161110"
            },
            "postId2" : {
                "title" : "Title2",
                "date" : "19901110"
            }
        }
    }
}
```

Por exemplo, como demonstrado na código 1, se tivéssemos um sistema de *blogs* e quiséssemos mostrar os *posts* por usuário poderíamos estruturar nosso banco de dados de forma que o nodo `post` tivesse como pai nodos com o `id` do

usuário como chave. Através disso, com o simples código abaixo, poderíamos acessar os posts de um usuário diretamente, sem precisar fazer uma busca entre todos os posts.

```
firebase.database().ref('post/${uid}').once();
```

Por esse motivo, é comum ouvir desenvolvedores descreverem o *Firebase* como um modelo de banco de dados orientado às necessidades da aplicação. Ou seja, independente das entidades que estamos lidando, construiremos nossa estrutura voltada para o que queremos mostrar ao usuário. Dessa forma, é necessário ter uma ideia geral de como a aplicação se comportará antes da modelagem do banco visto que, apesar da flexibilidade proporcionada pelo modelo, uma completa reestruturação ainda demandaria uma grande quantidade de trabalho.

Combinado a isso, o desenvolvedor pode configurar observadores que são objetos que ficam ativos no cliente e são informados caso houver qualquer alteração no banco de dados. Esta solução possibilita que o usuário tenha os dados sincronizados com o banco instantaneamente, sem precisar recarregar ou interagir com a aplicação para que sejam atualizados. Na seção 5.4 mostramos um experimento que busca medir a performance desse mecanismo de sincronização.

4.1.2 Segurança

Como já comentamos nesse capítulo, aplicações que usam a ideia de SPA tem um grande potencial para apresentar falhas de segurança. Assim, não basta termos uma conexão do cliente direto com um serviço se o serviço não proporcionar uma forma de restringir o acesso aos recursos - veremos com mais detalhes no experimento 5.2. Portanto, em uma arquitetura centrada no cliente, é papel do *BaaS* definir quem tem acesso de leitura e escrita. Além disso, também precisamos garantir a formatação dos dados salvos uma vez que, tendo acesso de escrita, o cliente pode tentar escrever informações em qualquer formato.

Com esse problema em mente, o *Firebase* projetou as *Security Rules* para descrever as regras do banco de dados. Com uma sintaxe estruturada em JSON, o desenvolvedor precisa arquitetar como ele limitará o acesso e a estrutura do banco. É importante ressaltar que as regras não descrevem a estrutura por completo, pois isso significaria perder a flexibilidade que um banco de dados baseado

em documentos proporciona. A intenção nesse caso é unicamente restringir pontos específicos do banco.

Código 2 Exemplo de Security Rules

```
{
    "rules": {
        "users": {
            "$userId": {
                ".read": "auth.uid === $userId",
                ".write": "auth.uid === $userId ||
                    !root.child('users').child($userId).exists()",
                "email": {
                    ".validate": "newData.isString()"
                },
                "name": {
                    ".validate": "newData.isString()"
                }
            }
        },
        "$other": {
            ".read": "true",
            ".write": "true"
        }
    }
}
```

As *Security Rules*, exemplificadas na código 2, definem o acesso ao banco através das primitivas `.read`, `.write` e `.validate`. A chave `rules` é obrigatória e o seu valor representa a estrutura do nodo raiz do banco. Dentro dela podemos declarar regras para os seus subnodos. Neste caso, estamos restringindo regras para o nodo `users` e deixando livre a criação e leitura de outros nodos através do caractere coringa `$`. Dentro de `users`, estamos usando o `$` novamente para mostrar que o banco deve aceitar qualquer valor como chave dos subnodos. Entretanto, através da regra de `read` estamos limitando que o cliente só terá acesso a esse nodo se estiver devidamente autenticado. Além disso, o usuário autenticado só terá acesso aos dados que o pertecem, ou seja, se ele tiver acessando a chave (`userId`) que é igual ao seu `ID` de usuário do objeto de autenticação (`auth.uid`) - entraremos em mais detalhes sobre a autenticação na próxima seção. Para possibilitar alterações e criação de novos usuários, a regra de `write` permite que o cliente escreva somente no subnodo do seu `ID` ou que crie esse caso não exista. Para finalizar, limitamos os campos de escrita em `email` e `nome do usuário` e definimos que estes devem ser *strings*.

Apesar das *Security Rules* serem uma ferramenta poderosa para a estruturação do banco, sua modelagem não é trivial e é comum chegar em situações onde precisamos remodelar o banco para alcançar o resultado desejado. O experimento da seção 5.3 exemplificará este tipo de situação.

Autenticação

Como apresentado anteriormente, é necessário um serviço de autenticação para garantir que o cliente que está alterando dados no banco é realmente quem ele se diz ser. Para isso, o Firebase possui um serviço de autenticação próprio que, apesar de não ter os detalhes de sua implementação expostos em suas documentações, buscaremos entender a partir da análise de sua *SDK*.

A autenticação é feita com a utilização de *JSON Web Tokens* (JWT) (JWT, 2016), que são retornados pelo *backend* depois que o usuário faz o *log-in* inicial. Esses tokens permitem a prática de *single sign-on* já que podem ser armazenados no browser do cliente através de *cookies* ou *local storage*. Assim, o usuário não precisa fornecer suas credenciais toda vez que acessa a aplicação.

Além disso os JWT's permitem o envio de informação de forma segura uma vez que os dados estão criptografados no token. É através desse recurso que o *Firebase* possibilita o uso do ID do usuário como um parâmetro para as *Security Rules*. O usuário fornece suas credenciais e o *Firebase* retorna um JWT com as informações relativas ao usuário. Assim, sempre que o cliente efetuar uma chamada ao *Firebase* o token será validado e as informações utilizadas para resolver as *Security Rules*.

4.2 A flexibilidade do Javascript

Vamos mostrar nessa seção como as Single Page Applications foram impulsionadas pela flexibilidade e desempenho que o Javascript provê devido aos paradigmas de programação por ele utilizado. Além do fato de trabalharmos com uma *Event Queue*, abordado na seção 2.2, a possibilidade de usar a orientação a objetos baseada em **composição de objetos** é essencial para o desenvolvimento em *Client Side Rendering*.

A necessidade de prover composição de objetos surgiu dos problemas do paradigma clássico de orientação a objetos. Uma vez que no paradigma clássico o desenvolvedor projeta suas classes baseado no que elas são e não no que elas fazem, é comum cair no problema de necessitar somente um método de uma classe mas ser obrigado a carregar toda a sua hierarquia. Em (SEIBEL, 2009), Joe Armstrong o descreve como o problema da banana e do gorila: "O problema de linguagens orientadas a objetos é que elas possuem todo esse ambiente que

elas carregam com elas. Você queria uma banana mas o que você consegue é um gorila carregando a banana e a floresta inteira”. Acrescentando, esse fato tem impulsionado uma crescente ideia de que a orientação a objetos clássica não deve ser utilizada sob nenhuma hipótese (ELLIOTT, 2014). Em seguida, vamos ver um exemplo para compreender a possível motivação para isso.

```
Usuario
    .comprarGasolina
    .trocarSenha
    .deletarConta

Veiculo
    .acelerar
    Carro
    Caminhao
```

Suponhamos que trabalhamos com um sistema que possui duas classes, **Usuário** e **Veículo**. Como mostrado acima, Veículo pode *acelerar* e ser da categoria ou Carro ou Caminhão. Usuário pode *comprarGasolina* e *trocarSenha*. Nesse caso, se quisermos introduzir um Carro Autônomo que pode *comprarGasolina* começamos a ter problemas. Uma vez que não queremos duplicar código, a alternativa seria criar uma classe pai a Usuário e Veículo com o método em comum entre Carro e Usuário.

```
Comprador
    .comprarGasolina

    Usuario
        .trocarSenha
        .deletarConta

    Veiculo
        .acelerar
        Carro
        Caminhao
        CarroAutonomo
```

A estrutura acima possibilitaria construir um Carro Autônomo da forma como esperávamos, porém alguns dos objetos teriam funcionalidades que eles não usariam. Nesse exemplo, Caminhão e Carro não são autônomos e portanto não podem *comprarGasolina*, mesmo assim sua classe pai possui esse método e, portanto, ele será carregado sempre que um novo objeto Caminhão ou Carro for instanciado.

Esse problema é facilmente contornado usando Composição de Objetos, o qual é de simples utilização no Javascript através do método `Object.assign()`. Primeiramente, precisamos definir algumas *factories* (ELLIOTT, 2016) com os métodos que serão designados:

```
const compra = (estado) => ({
    comprarGasolina: () =>
        console.log(`gasolina comprada por ${estado.nome}`);
});

const acelera = (estado) => ({
        acelerar: () =>
        console.log(`${estado.nome}
            acelerando a 88 milhas por hora`);
});

const gerenciaConta = (estado) => ({
        trocarSenha: () =>
        console.log(`senha de ${estado.nome} alterada`);
        excluirConta: () =>
        console.log(`conta de ${estado.nome} excluida`);
});
```

Uma vez que temos todos os métodos definidos, só precisamos designar quais métodos cada objeto deve possuir. Portanto, o que temos são as seguinte estruturas:

A partir disso, criamos um construtor para os objetos e, assim, podemos construir objetos facilmente, como mostramos no código 3.

Vimos que através de composição de objetos o Javascript disponibiliza uma maneira muito mais leve e flexível de estruturar aplicações. Uma vez que

Table 4.1: Objetos e suas ações

Objetos	**Ações**
Carro	acelera
Caminhão	acelera
Usuário	compra e gerenciaConta
Carro Autonomo	compra e acelera

Código 3 Compondo o Objeto

```
const carroAutonomo = (nome) => {
    let estado = {
        nome,
        rodas: 4
    };
    return Object.assign(
        {},
        acelera(estado),
        compra(estado)
    );
}

carroAutonomo('Lada').acelerar();
//Lada acelerando a 88 milhas por hora
```

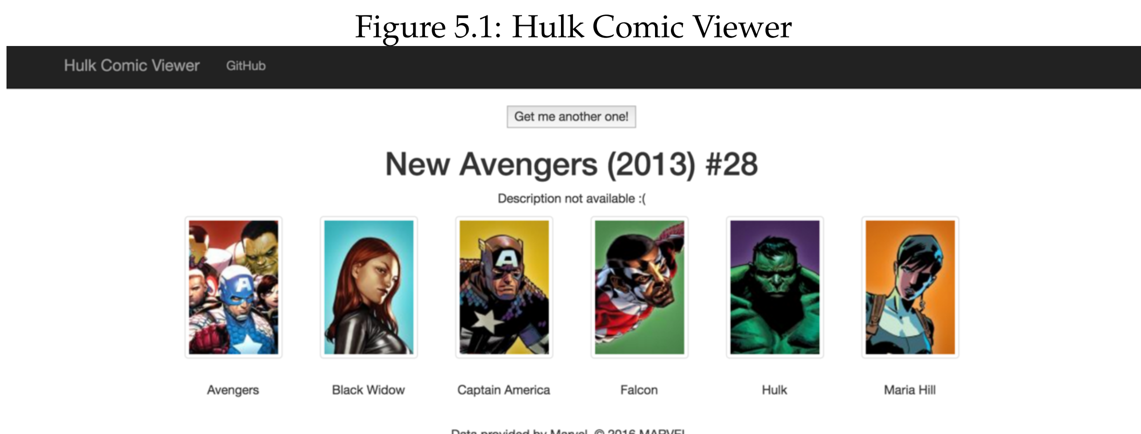
não podemos abusar de recursos em aplicações que rodam no navegador, tal característica se torna essencial para possibilitar esse tipo de arquitetura.

5 EXPERIMENTOS

Passamos pelos capítulos 3 e 4 falando de diferentes abordagens de arquiteturas web de forma conceitual. Nesse capítulo, iremos descrever 4 experimentos que planejamos e realizamos para avaliar na prática o que estudamos até aqui e, assim, poder argumentar adequadamente na escolha da arquitetura a adotar no desenvolvimento de uma aplicação web.

5.1 Experimento 1 : Hulk Comic Viewer

Figure 5.1: Hulk Comic Viewer



Nosso primeiro experimento será baseado em uma aplicação para visualizar informações de revistas em quadrinhos da *Marvel* [1]. Para isso, foi utilizada a API da empresa (MARVEL, 2016), que fornece dados sobre as histórias e personagens em formato JSON. Como podemos ver na figura 5.1, a aplicação provê o título do quadrinho, uma descrição (se esta estiver disponível), e a lista de personagens que participam da história com nome e foto. Toda vez que o usuário aperta o botão *Get me another one!*, a aplicação carrega randomicamente um novo quadrinho e suas respectivas informações.

A primeira chamada para a API retorna um JSON descrevendo o quadrinho randomico desejado, com o título, descrição e referências aos personagens, como exemplificado em 4. Observe que esse objeto nos fornece apenas lista dos personagens e seus identificadores, sem as suas imagens. Portanto, é necessário usar

[1] https://github.com/adolfosrs/hulk-comic-viewer/tree/master

esses identificadores para fazer uma chamada para cada personagem para que possamos ter acesso suas imagens. Dessa forma, o número de requisições feitas à API da *Marvel* sempre será um mais o número de personagens do quadrinho. Neste experimento, iremos comparar a performance da aplicação entre uma solução utilizando *Server Side Rendering* sem *Ajax*, uma introduzindo o *Ajax* e, finalmente, uma solução baseada em *Client Side Rendering*.

Código 4 Marvel JSON

```
{
  "data": {
    "code": 200,
    "status": "Ok",
    "copyright": "© 2016 MARVEL",
    "attributionText": "Data provided by Marvel. © 2016 MARVEL",
    "attributionHTML": "<a href=\"http://marvel.com\">© 2016 MARVEL</a>",
    "data": {
      "results": [
        {
          "id": 18253,
          "title": "Here Comes The Hulk!",
          "description": "",
          "resourceURI": "http://gateway.marvel.com/v1/public/stories/18253",
          "characters": {
            "available": 2,
            "collectionURI": "http://gateway.marvel.com/v1/public/stories/18253/characters",
            "items": [
              {
                "resourceURI": "http://gateway.marvel.com/v1/public/characters/1009224",
                "name": "Captain Marvel (Mar-Vell)"
              },
              {
                "resourceURI": "http://gateway.marvel.com/v1/public/characters/1009351",
                "name": "Hulk"
              }
            ],
            "returned": 2
          },
  "statusText": "OK"
}
```

5.1.1 Server Side Rendering sem Ajax

Uma vez que não estamos considerando utilizar recursos do cliente nessa situação, temos que resolver todas as chamadas no lado do servidor. Portanto, utilizaremos *Ruby on Rails* sem o auxílio do Javascript. Nosso `HTML` possui marcações, como mostrado no *snippet* 5, que são resolvidas com a execução do código *Ruby*. Após o processamento de todas as marcações, o servidor *rails* responde ao cliente com o `HTML` já renderizado.

Primeiramente, o cliente faz a chamada para o servidor da aplicação e este efetua a primeira chamada para a API da Marvel. Então, aguarda a resposta con-

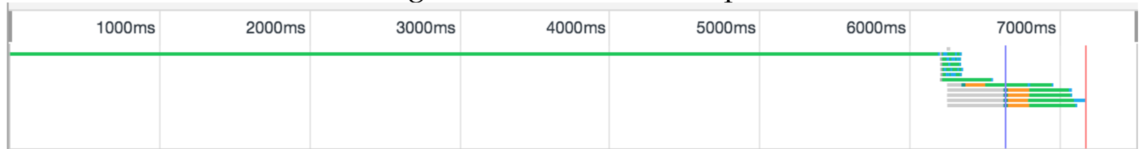
Código 5 Marcações Hulk

```
<h1><%= @comic.title %></h1>
```

tendo os dados da revista e, em seguida, para cada um dos personagens da revista em questão, efetua uma nova chamada de forma síncrona, sempre esperando o retorno de uma para requisitar a próxima.

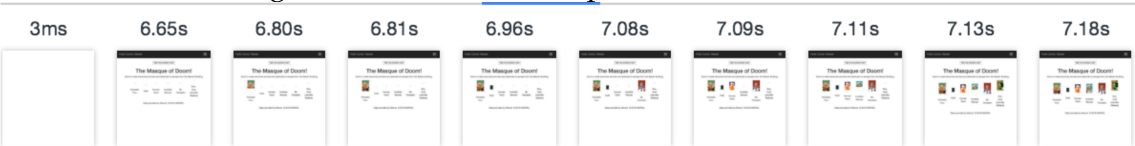
Resultados: Neste caso, o resultado dependerá da revista que recebemos, pois precisamos esperar o servidor fazer as chamadas para cada um dos personagens da história.

Figure 5.2: Linha de tempo



Nas figuras 5.2 e 5.3 podemos observar mais detalhadamente o carregamento da página em função do tempo. A figura 5.2 demonstra um gráfico com o tempo em milissegundos no eixo X no qual a linha verde demonstra o tempo em que o navegador está esperando uma requisição ao servidor. A figura 5.3 mostra cada um dos frames da aplicação até que ela seja completamente carregada, e quanto tempo, em segundos, demorou para cada frame ser disponibilizado ao usuário. Ambos os gráficos são gerados pelas ferramentas de desenvolvedor do navegador Google Chrome (GOOGLE, 2016). Neste experimento, estamos carregando uma revista com 6 personagens e, observando a figura 5.2, fica nítido que o navegador está permanecendo muito tempo esperando a resolução da chamada do servidor. São 6.3 segundos de espera para receber a resposta e iniciar o *download* dos primeiros dados. Por esse motivo, vemos na figura 5.3 que o primeiro frame só é disponibilizado após 6.65 segundos de execução.

Figure 5.3: Frames do experimento server side



Complementando, se observarmos os tempos de execução dentro do servidor, pelo uso de logs no código da aplicação, é possível detectar que o tempo de

efetuar a chamada dos dados da revista e receber a resposta leva, em média, 1.43 segundos. Já as chamadas para acessar o endereço da imagem de cada personagem levam em média 0.52 segundos.

No próximo experimento, veremos como a utilização de Ajax pode diminuir o tempo ocioso do navegador e melhorar a experiência do usuário.

5.1.2 Server Side Rendering com Ajax

O objetivo desse experimento é mostrar como é possível melhorar a performance de nossa aplicação utilizando as ferramentas de assincronia do Javascript aplicadas com o auxílio do Ajax. Portanto, nesse exemplo [2] chamamos o servidor apenas para obter os dados da revista e deixamos as chamadas para acessar o endereço das imagens para serem feitas no navegador do usuário.

Resultados: As figuras 5.4 e 5.5 mostram detalhadamente como o tempo ocioso do navegador diminuiu drasticamente. Na figura 5.4, vemos que a primeira chamada retorna após aproximadamente 1.3 segundos. Depois disto, o *download* dos recursos de CSS e Javascript da página é iniciado. Após ter baixado o Javascript da página, o navegador vai, utilizando Ajax, requisitar as imagens dos personagens de forma assíncrona. Assim, observamos na figura 5.5 que o primeiro frame já é disponibilizado com 1.81 segundos de execução e as imagens foram requisitadas alguns instantes antes disto. Todas as imagens tiveram suas chamadas disparadas simultaneamente e estas são apresentadas para o usuário conforme o servidor começa a responder. Neste caso, aos 2.28 segundos de execução já temos a primeira imagem e aos 2.77 todas as imagens já estão carregadas.

Figure 5.4: Linha do tempo server side com ajax

[2] https://github.com/adolfosrs/hulk-comic-viewer/tree/async

Figure 5.5: Frames do experimento server side com ajax

14ms 1.81s 2.28s 2.40s 2.47s 2.49s 2.52s 2.77s

5.1.3 Client Side Rendering

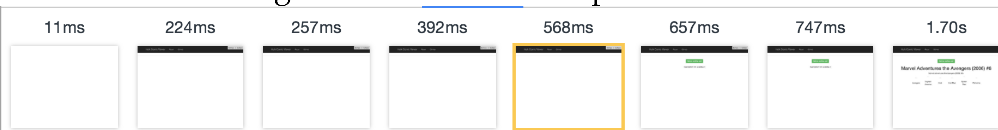
Através da abordagem client side [3] podemos, assim como na abordagem *Server Side* utilizando Ajax, requisitar as fotos dos personagens de forma assíncrona. Entretanto, como não estamos fazendo requisições a um servidor intermediário, teremos um Round Trip Time (RTT) a menos por chamada. Isso é, teremos menos tempo de propagação de dados. Além disso, podemos navegar pelas diferentes páginas da aplicação enquanto esperamos as requisições, uma vez que não recarregamos a aplicação quando trocamos de página.

Resultados: Como estamos utilizando uma abordagem de renderização da página no cliente, temos um tempo de resposta ao usuário muito mais rápida. Como vemos na figura 5.7, o primeiro elemento da página é apresentado em somente 224 milissegundos. Ao analisarmos a figura 5.6 vemos que, ao contrário do qua ainda tínhamos na solução utilizando Ajax, não temos uma grande espera na chamada inicial ao servidor. Assim, conseguimos baixar os elementos CSS da página quase instantaneamente e possibilitar uma melhor experiência ao usuário uma vez que esse rapidamente possui conteúdo para analisar e interagir. Portanto, o resultado final é uma aplicação rápida e fluida.

Figure 5.6: Linha do tempo client side

[3] https://github.com/adolfosrs/hulk-comic-viewer/tree/angular

Figure 5.7: Frames do experimento client side



5.2 Experimento 2: Segurança client side

Ter uma aplicação web centrada no cliente significa disponibilizar o código ao navegador e, consequentemente, ao usuário final. Esse fato abre portas para diferentes formas de problemas de segurança. Neste experimento veremos algumas delas.

Iremos explorar o *Babies Book* [4], uma aplicação voltada para pais que desejam postar as fotos de seus filhos na internet sem a grande exposição intrínseca das redes sociais mais tradicionais. Ao accessar a página usando o *Chrome*, podemos abrir as ferramentas de desenvolvedor e ver o `HTML` e outros arquivos que o navegador precisou carregar para iniciar a aplicação. Ao inspecionar o `HTML`, é possível ver todos as bibliotecas que a página está importando. Entre elas, como exemplificado no código abaixo, o Angular [5], um framework de desenvolvimento de SPA.

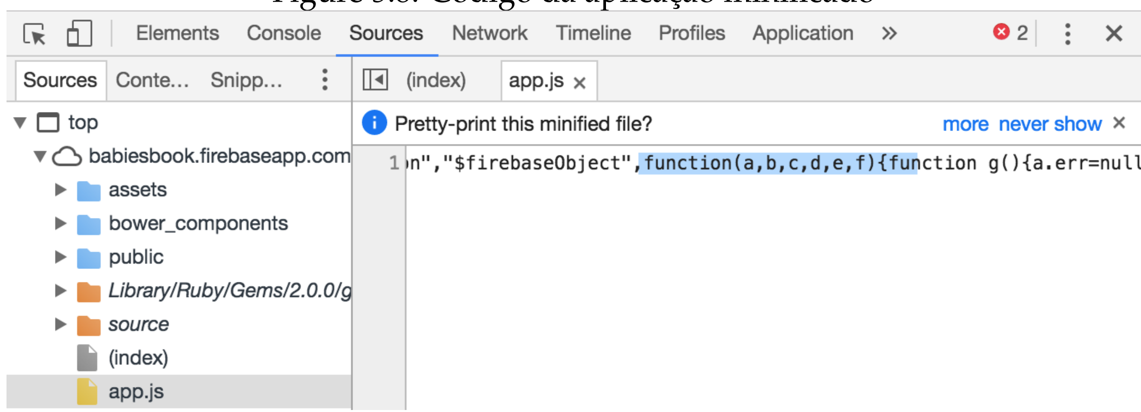
```
<script src="bower_components/angular/angular.js"></script>
```

Ao procurar por arquivos de Javascript, encontramos o `app.js`, que contém toda a lógica de aplicação. Percebemos que esse código está usando a técnica de minificação (HANSELMAN, 2011) com o intuito de tornar o código ilegível e diminuir o seu tamanho - figura 5.8. Essa técnica dificulta o entedimento do código de programação. Entretanto, o código em si não possui informações sensíveis e sua anonimidade não é crucial para a segurança da aplicação. Observando essa página e seus recursos, podemos concluir que o arquivo `app.js` contém todo o código que a aplicação utiliza e, considerando que ela se baseia em dados disponibilizados por outro usuários, é evidente que existem conexões com outros servidores. São essas conexões que trazem os maiores riscos de segurança, visto que são comunicações com serviços externos que geralmente armazenam dados de usuários, que podem variar desde email pessoal até conta bancária. No

[4] http://babiesbook.firebaseapp.com
[5] https://angularjs.org/

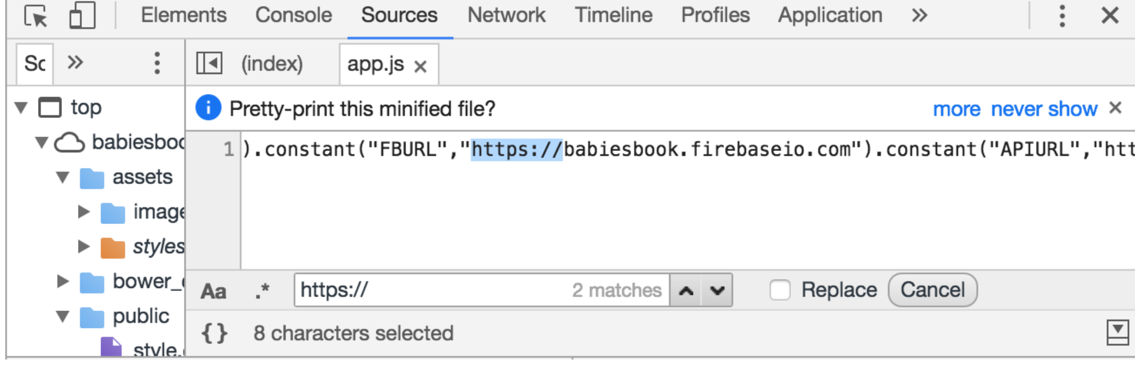
caso do *Babies Book*, garantir que os dados do usuário estão protegidos é essencial para o sucesso do produto, já que ele se apresenta como uma plataforma restrita e segura para o compartilhamento de informações da família.

Figure 5.8: Código da aplicação minificado



Fazendo uma busca textual por `https://`, encontramos quais são os outros endereços para os quais a página efetua chamadas e temos acesso às URLs dos servidores que a aplicação utiliza. Na figura 5.9, vemos que o Firebase está sendo utilizado e que o link do servidor é `https://babiesbook.firebaseio.com`.

Figure 5.9: Procurando servidores acessados pela aplicação



Tendo alguns conhecimentos básicos de como o *Firebase* funciona saberemos que o link em questão é suficiente para disparar uma chamada no banco de dados. Utilizando o console do *Chrome* podemos utilizar o comando `new Firebase('https://babiesbook.firebaseio.com').once()` para fazer uma chamada no diretório raiz do banco, o que nos daria acesso a todas as informações da aplicação - figura 5.10.

Além disso, como mostrado na figura 5.11, poderíamos fazer alterações no banco, utilizando o comando a seguir:

Figure 5.10: Dados da aplicação acessíveis

```
Elements  Console  Sources  Network  Timeline  Profiles  Application  »

top  ▼  Preserve log

> new Firebase('https://babiesbook.firebaseio.com').once('value', snapshot =>
  console.log(snapshot.val()))
<· ▶Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}
  ▼Object                                                            VM1750:1
    ▼babies: Object
      ▶adolfosrs: Object
      ▶adolfotetet: Object
      ▼jorge: Object
         createdAt: "Fri May 20 2016 09:10:33 GMT-0300 (BRT)"
         gender: "Male"
         isPrivate: false
         name: "Jorge"
         owner: "67031711-e8c0-4df5-93a1-70029c2df940"
         profileImgName: "default-icon@2x.png"
        ▶relatives: Object
        ▶__proto__: Object
      ▶magalzao: Object
      ▶teio: Object
      ▶thom: Object
      ▶__proto__: Object
    ▶events: Object
    ▶posts: Object
    ▶sizes: Object
    ▶users: Object
    ▶usersProfile: Object
    ▶__proto__: Object
```

```
new Firebase('https://babiesbook.firebaseio.com').set()
```

Figure 5.11: Dados da aplicação alteráveis

```
top  ▼  Preserve log

> new Firebase('https://babiesbook.firebaseio.com').child('babies/jorge/name').set('Jorge
  Costa')
<· ▶Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}
> new Firebase('https://babiesbook.firebaseio.com').child('babies/jorge/name').once('value',
  snapshot => console.log(snapshot.val()))
  Jorge Costa                                                        VM2080:1
<· ▶Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: W}
```

Vimos no capítulo 4 que problemas de segurança como este são consequência desta nova arquitetura de desenvolvimento web. Vimos também que ferramentas são criadas constantemente para suportar esse novo paradigma e que com elas surgem soluções para estes novos problemas encontrados. No caso do Firebase, existem as regras de segurança (*Security Rules*), apresentadas na seção 4.1.2.

As *Security Rules* do *Babies Book* foram momentaneamente desativadas para fins de análise desse experimento e elas podem ser encontradas do apêndice A.

Caso estivessem ativas, o cliente, ao tentar acessar dados restritos, receberia a mensagem mostrada na figura 5.12.

Figure 5.12: Firebase - Acesso Negado

```
▶Error: permission_denied at /posts/jorge: Client doesn't have permission to access the desired   angular.js:13550
data.
    at I (firebase.js:217)
    at Object.I (firebase.js:220)
    at firebase.js:184
    at Rh.h.Ld (firebase.js:189)
    at Fh.Ld (firebase.js:179)
    at wh.Lg (firebase.js:177)
    at zh (firebase.js:171)
    at WebSocket.La.onmessage (firebase.js:170)
```

Essas regras definem a estrutura que o banco de dados vai ter, portanto é crucial entender como elas se comportam para que a aplicação seja performática e segura. No próximo experimento, veremos alguns erros comuns e boas práticas para a modelagem com o Firebase.

5.3 Experimento 3: Complexidade das Security Rules

Nesse experimento vamos mostrar como as *Security Rules* do Firebase afetam diretamente a modelagem do banco de dados. Sua sintaxe rígida resulta em uma curva de aprendizagem longa e, muitas vezes, em contínuas remodelagens da estrutura do banco. Iremos analisar, nesse caso, a regra dos *posts* para a aplicação *Babies Book*. Os posts dessa aplicação se comportam da mesma forma que os do Facebook: podem conter fotos, textos, vídeos, entre outros, e ficam disponibilizados em uma timeline cronológica na qual o mais recente aparece no topo.

Desenvolvedores iniciantes tendem a modelar o nodo *posts* de forma que este seja composto por subnodos que contenham uma propriedade apontando para o ID do bebê o qual a postagem se refere. A estrutura das regras nesse caso se pareceriam com o que vemos na código 6.

Uma vez que queremos possibilitar privacidade para as postagens teremos uma opção para tornar todo o conteúdo do bebê privado. Para fins de análise teremos uma *flag* `isPrivate` contida no nodo do bebê que indicará se as postagens podem ser visualizadas. O erro comum nesse ponto é esquecer que as regras não funcionam como filtros e montar a regra de leitura em `$postId` como mostrado no código 7.

Explicando, a regra avaliará para positivo se o post que está sendo acessado (`$postId`) pertencer a um ID de bebê (`data.child('babyID').val()`)

Código 6 Modelagem Security Rules - ID como propriedade

```
{
    "posts": {
        "$postId": {
            ".read": "true",
            "babyID": {
                ".validate": "newData.isString()"
            },
            "content": {
                ".validate": "newData.isString()"
            }
        }
    }
}
```

Código 7 Exemplo de erro - Regras não são filtros

```
{
    ".read":
        "root.child('babies').child(data.child('babyID').val())
            .child('isPrivate').val() === false"
}
```

o qual tenha o valor da *flag* `isPrivate` como `false` em `/babies/$babyID`.

O problema é que as regras são atômicas, isso é, só avaliam o acesso ao nodo requisitado. Em caso de termos acesso, todos os subnodos serão visíveis, caso contrário o firebase responderá com uma mensagem de accesso negado e nenhum dado será mostrado. Na aplicação em questão queremos buscar os posts dado um ID de um bebê. A requisição que necessitamos é a seguinte:

```
firebase.database().ref('posts').orderByChild('babyID')
    .equalTo(babyID).once();
```

Entretanto nesse caso estamos filtrando pelos posts que nos interessam acessando o nodo `posts`. Visto que não construimos regras para esse nodo o acesso será avaliado para falso e portanto não conseguiremos o comportamento esperado. A estrutura de regra que temos até agora só funcionaria se tivéssemos os IDs dos posts que quiséssemos acessar e assim acessaríamos o nodo de cada post diretamente, como mostrado abaixo.

```
firebase.database().ref(`posts/${postID}`).once()
```

Nesse caso não temos outra escolha se não remodelar nossso banco. Nossa aplicação é centrada em apresentar os *posts* de cada bebê por vez, portanto podemos separar os posts pelo ID do bebê (`/posts/$babyID/$postID`) - ver código 8. Essa abordagem facilitará a construção das regras e, como vimos na seção 4.1.1 nos proporcionará uma melhor performance.

Código 8 Modelagem Security Rules - ID como chave

```
{
    "posts": {
        "$babyID": {
            ".read": "root.child('babies').child($babyID)
                .child('isPrivate').val() === false",
            "$postID": {
                "content": {
                    ".validate": "newData.isString()"
                }
            }
        }
    }
}
```

Sendo que iremos acessar as postagens através do subnodo `/posts/$babyID` não enfrentaremos o problema que tivemos anteriormente. Com a chamada mostrada na código 9 passaremos pela regra que definimos. Assim, será verificado se o bebê em questão possui a *flag* ativa ou não e, caso negativo, finalmente teremos acesso aos respectivos posts.

Código 9 Chamada direta às postagens do bebê

```
firebase.database().ref(`posts/${babyID}`).once()
```

5.4 Experimento 4: Performance banco de dados em tempo real

O *Firebase* se apresenta como um banco de dados que atualiza as informações em tempo real em todos os dispositivos que estiverem acessando estas informações. Neste experimento, verificaremos a performance desse serviço, analisando o tempo de resposta entre a alteração de um dado e sua respectiva propagação para os dispositivos que o estão observando. Nosso objetivo é quantificar o que a solução chama de tempo real.

A sincronização dos dados é feita através dos observadores disponibilizados pela *SDK*. Com o código abaixo, o desenvolvedor pode configurar um observador em um nodo do banco, onde `event_type` é o tipo de evento que a aplicação irá observar. Sempre que o evento especificado acontecer, o *Firebase* irá avisar os observadores e o método de *callback* será disparado.

```
firebase.database().ref(users).on(event_type, callback);
```

Para testarmos o tempo entre a atualização do dado e ter o *callback* do observador chamado, inserimos dados em um nodo e começamos a observar o evento `child_added` que é disparado sempre que um novo subnodo é adicionado. A medição foi feita salvando um *timestamp unix* no instante em que enviamos o request para a inserção do subnodo e posteriormente comparando esse com um novo *timestamp* gerado no *callback* do observador. O tamanho final do objeto salvo em cada inserção é de 16 bytes.

Adicionando um único subnodo por chamada ao firebase conseguimos, em uma amostra de 10000 chamadas, uma média de 200.23 milissegundos de tempo de resposta. O menor tempo da amostragem foi de 75 milissegundos e o maior 1.45 segundos, causado muito provavelmente por instabilidade da conexão. O intervalo de tempo escolhido entre uma inserção e outra foi de 1 segundo.

6 CONCLUSÃO

Nesse trabalho, nós defendemos o uso do modelo de Client Side Rendering através das SPAs apresentando os problemas de performance e experiência do usuário das soluções de Server Side Rendering. Através de um experimento, foi possível quantificar o tempo ocioso que o usuário enfrenta em aplicações renderizadas no servidor e entender como diminiuir esse tempo delegando tarefas ao cliente. Vimos que o Ajax, combinado com Server Side Rendering, é capaz de resolver muitos dos problemas descritos, mas que o uso das SPAs ainda é superior. Nas SPAs, não precisamos recarregar a página em nenhuma interação e, no primeiro carregamento, conseguimos apresentar elementos de tela ao usuário quase instantaneamente. Vimos também a importância do Javascript para proporcionar aplicações que rodam inteiramente no cliente. Através da composição de objetos, uma característica do Javascript, conseguimos aplicações mais leves e flexíveis.

Apesar dos benefícios relacionados à experiência do usuário, identificamos que o uso de Client Side Rendering deve ser acompanhado de precauções de segurança. Além disso, defendemos o uso de serviços que entreguem a performance adequada para esse tipo de aplicação. Para alcançar tais fatores, recomendamos o uso de BaaS especializados nessa arquitetura. Dentre esses serviços, exploramos o Firebase, mostrando como ele resolve os problemas levantados e que, apesar de sua abordagem simples, estruturar o banco e suas respectivas regras de segurança não são tarefas triviais.

Procuramos trabalhar a nossa abordagem de forma mais genérica, sem entrar em detalhes das diferentes soluções de SPA e BaaS. Essa decisão foi tomada pois, como mencionamos no capítulo 1, estas soluções estão em constante transformação e um trabalho focado em alguma delas se tornaria obsoleto em um curto período de tempo. Portanto, consideramos mais relevante apresentar uma visão geral dessa arquitetura, as motivações para migrar o processamento para o lado do cliente e como combinar diferentes soluções para construir aplicações mais rápidas, leves e flexíveis.

Finalmente, acreditamos que as SPAs são um grande passo para um futuro de desenvolvimento web com processamento totalmente descentralizado. Em trabalhos futuros, é interessante estudar a viabilidade de descentrar não so-

mente a aplicação frontend através das SPAs, mas ter, combinado a essas, serviços de backend que executem da mesma forma. Uma variedade desses serviços está surgindo com as redes Blockchain (CROSBY et al., 2015) e, mais especificamente, plataformas como a Ethereum [1], onde desenvolvedores podem rodar seus códigos em uma rede peer-to-peer que utiliza criptografia para garantir que o código não seja adulterado. Um exemplo desse tipo de serviço é o Storj [2] (WILKINSON, 2014), que permite a armazenagem , de forma segura, de qualquer tipo de arquivo em uma rede descentralizada.

[1]https://www.ethereum.org/
[2]https://storj.io/

BIBLIOGRAPHY

AMAZON. **Amazon EC2 – Hospedagem de servidor virtual**. 2016. <https://aws.amazon.com/pt/ec2/>. Accessed: 2016-11-16.

CROSBY, M. et al. **BlockChain Technology: Beyond Bitcoin**. 2015. <http://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf>. Accessed: 2016-11-22.

ELLIOTT, E. **The Two Pillars of JavaScript - Part 1: How to Escape the 7th Circle of Hell**. 2014. <https://medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3>. Accessed: 2016-11-22.

ELLIOTT, E. **JavaScript Factory Functions vs Constructor Functions vs Classes**. 2016. <https://medium.com/javascript-scene/javascript-factory-functions-vs-constructor-functions-vs-classes-2f22ceddf33e#.lqxvryk74>. Accessed: 2016-11-22.

GARRETT, J. J. **Ajax: A New Approach to Web Applications**. 2005. <http://adaptivepath.org/ideas/ajax-new-approach-web-applications/>. Accessed: 2016-11-13.

GOOGLE. **Chrome DevTools Overview**. 2016. <https://developer.chrome.com/devtools>. Accessed: 2016-11-14.

GRAVELLE, R. **Introducing HTML 5 Web Workers: Bringing Multi-threading to JavaScript**. 2016. <http://www.htmlgoodies.com/html5/tutorials/introducing-html-5-web-workers-bringing-multi-threading-to-javascript.html#fbid=Vy2extSCMqf>. Accessed: 2016-11-19.

HANSELMAN, S. **The Importance (and Ease) of Minifying your CSS and JavaScript and Optimizing PNGs for your Blog or Website**. 2011. <http://www.hanselman.com/blog/TheImportanceAndEaseOfMinifyingYourCSSAndJavaScriptAndOptimizingPNGsForYourBlogO aspx>. Accessed: 2016-11-22.

JR., K. S. **Load Balancing 101: Nuts and Bolts**. 2012. <https://f5.com/resources/white-papers/load-balancing-101-nuts-and-bolts>. Accessed: 2016-11-16.

JWT. **Introduction to JSON Web Tokens**. 2016. <https://jwt.io/introduction/>. Accessed: 2016-11-21.

LANGE, Y. de. **Single Page Application considerations**. 2016. <https://stovepipe.systems/post/single-page-application-considerations>. Accessed: 2016-11-14.

MARTENSEN, D. **Events, Concurrency and JavaScript**. 2015. <https://danmartensen.svbtle.com/events-concurrency-and-javascript>. Accessed: 2016-11-22.

MARVEL. 2016. <https://developer.marvel.com/>. Accessed: 2016-09-28.

MITCHELL, B. **Introduction to Client Server Networks**. 2016. <https://www.lifewire.com/introduction-to-client-server-networks-817420>. Accessed: 2016-11-16.

SEIBEL, P. **Coders at Work: Reflections on the Craft of Programming**. [S.l.]: Apress, 2009.

WILKINSON, S. **Storj, A Peer-to-Peer Cloud Storage Network**. 2014. <https://storj.io/storj.pdf>. Accessed: 2016-11-22.

APPENDIX A — REGRAS DE SEGURANÇA DO BABIES BOOK

```
{
   "rules": {
      "users": {
         "$user": {
            ".read": "auth.uid === $user",
            ".write": "auth.uid === $user || !root.child('users').child($user).exists()",
            "fullName": {
              ".validate": "newData.isString() && newData.val().length <= 2000"
            },
            "email": {
              ".validate": "newData.isString() && newData.val().length <= 2000"
            },
            "facebookId": {
            },
            "babies": {
            },
            "gender": {
            },
            "locale": {
            },
            "$other": {
               ".validate": true
            }
         }
      },
      "usersProfile": {
          ".indexOn": "createdAtDesc",
          "$userProfile": {
            ".write": "auth.uid === $userProfile
                       || !root.child('usersProfile').child($userProfile).exists()",
            ".read": true,
            "$other": {
               ".validate": true
            }
         }
      },
      "babies": {
         "$baby": {
            ".write": "!root.child('babies').child($baby).exists() ||
                        (root.child('babies').child($baby).child('relatives').hasChild(auth.uid))",
            ".read": "(data.child('isPrivate').val() === false)  ||
                           data.child('relatives').hasChild(auth.uid)",
            "createdAt":{
            },
            "gender": {
            },
            "isPrivate": {
            },
            "relatives": {
```

```
          "$userId":{
          }
        },
        "profileImgName": {
        },
        "name": {
          ".validate": "newData.isString() && newData.val().length <= 2000"
        },
        "owner": {
          ".validate": "(newData.isString() && newData.val().length <= 2000)
                        || data.val() === newData.val()"
        },
        "$other": {
           ".validate": true
        }
      }
    },
    "posts": {
        ".read": true,
        ".write": "auth != null",
        "$baby": {
          ".indexOn": "createdAtDesc",
          "$post": {
            ".read": "(root.child('babies').child(data.child('ownedBy').val()).child('isPrivate').val()
                    === false)  || root.child('babies').child(data.child('ownedBy').val())
                        .child('relatives').hasChild(auth.uid)",
            ".write": "!data.exists()
                     && (root.child('babies').child(newData.child('ownedBy').val()).exists()
                     && root.child('babies').child(newData.child('ownedBy').val())
                            .child('relatives').hasChild(auth.uid))",
            ".validate": "auth.uid === newData.child('postedBy').val()",
            "postedBy": {
              ".validate": "newData.isString() && root.child('users').child(newData.val()).exists()"
            },
            "ownedBy": {
              ".validate": "newData.isString() && root.child('babies').child(newData.val()).exists()"
            },
            "$other": {
               ".validate": true
            }
          }
        }
    },
    "events": {
        ".read": true,
        ".indexOn": "createdAt",
        "$post": {
          ".read": true,
          ".write": true,
          "$other": {
             ".validate": true
          }
```

```
        }
      },
      "sizes": {
          ".read": true,
          ".indexOn": "createdAt",
          "$post": {
            ".read": true,
            ".write": true,
            "$other": {
               ".validate": true
            }
        }
      }
    }
}
```