UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DOUGLAS JUNQUEIRA

DESENVOLVIMENTO DE *SOFTWARE* USANDO ANGULAR E NODE PARA ASSISTÊNCIA SOCIAL

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2018

DOUGLAS JUNQUEIRA

DESENVOLVIMENTO DE *SOFTWARE* USANDO ANGULAR E NODE PARA ASSISTÊNCIA SOCIAL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Richard Duarte Ribeiro

PONTA GROSSA

2018

TERMO DE APROVAÇÃO

DESENVOLVIMENTO DE *SOFTWARE* USANDO ANGULAR E NODE PARA ASSISTÊNCIA SOCIAL

por

DOUGLAS JUNQUEIRA

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 14 de Junho de 2018 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Dr. Richard Duarte Ribeiro
Orientador(a)

_____
Prof. Dr. Diego Roberto Antunes
Membro titular

_____
Prof. MSc. Rafael dos Passos Canteri
Membro titular

_____
Prof(a). Dra. Helyane Bronoski Borges
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. MSc. Saulo Jorge Beltrão de Queiroz
Coordenador do curso

RESUMO

JUNQUEIRA, D. *Desenvolvimento de software usando Angular e Node para assistência social*. 2018. 65 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) — Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

O desenvolvimento de sistemas para *web*, que sejam escaláveis e eficientes não é apenas uma tarefa difícil, mas um desafio a medida que a demanda, disponibilidade por funções e recursos aumenta. Além do desafio de desenvolvimento, existem muitas organizações que ainda atuam com base em papel e caneta devido ao custo desses sistemas para construção e manutenção. Assim, este trabalho visa apresentar os conceitos usados no mercado, para o desenvolvimento de arquiteturas voltadas a web, desde os sistemas clássicos à orientação a eventos, além da construção desenvolvido em Angular para a aplicação cliente, Node para servidor e SQL Server, através da licença fornecida, como banco de dados oferecem uma solução poderosa e completa. Os resultados avaliados não apresentam o cenário ideal, mas demonstram a viabilidade do uso em redes locais. Através deste trabalho conclui-se que o sistema pode ser usado para atender o objetivo principal e adicionais previstos no trabalho.

Palavras-chaves: *Framework*. Node. Angular. SQL Server. Social. Assistência.

ABSTRACT

JUNQUEIRA, D. *Software development using Angular and Node framework for Social Assistance*. 2018. 65 f. Work of Conclusion Course (Undergraduation in Computer Science) — Federal University of Technology - Paraná. Ponta Grossa, 2018.

Developing web-based systems that are scalable and efficient is not an easy task but rather a challenge as demand of access, data availability, and functions increases. In addition to the development challenge, there are many organizations that act on pen and paper because of the cost for construction and maintenance of web systems. Thus, this work aims to present the concepts used in the market, for the development of web-oriented architectures, from classic methods to event-driven systems, in addition to the developed in Angular for application client, Node for the server and SQL Server, through a provided license, as a database offers a powerful and complete solution, but easy to develop and maintain, as an accessible solution to social assistance organizations. The results do not represent the ideal scenario, but demonstrate a viability of the use in local networks. As result, the system can be used to meet the main and additionally provided objectives of this article.

Key-words: Node. Angular. SQL Server. Social. Assistance.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CPU	Unidade Central de Processamento
CRUD	*Create, Read, Update e Delete*
DOM	Modelo de Objeto de Documento
E/S	Entradas e Saídas
EV	*Event-Loop*
HTML	Linguagem de Marcação de Hipertexto
HTTP	Protocolo de Transferência de Hipertexto Seguro
Jocum	Jovens com Uma Missão
Jocum PG	Jocum Ponta Grossa
JSON	JavaScript Object Notation
MVC	*Model View Controller*
Node	Node.js
RAM	Random Access Memory
REST	Transferência de Estado Representacional
SGBD	Sistema Gerenciador de Banco de Dados
SO	Sistema Operacional
SSD	Unidade de estado sólido
SUAS	Sistema Único de Assistência Social
TI	Tecnologia da Informação
URL	Localizador Padrão de Recursos

SUMÁRIO

1 INTRODUÇÃO

Um dos maiores problemas sociais, que afeta a população mundial, é a pobreza. Em 2013, um estudo mostrou que até 4 bilhões de pessoas vivem com até R$ 8,00 para manutenção diária, não limitado à apenas países em desenvolvimento (CHOWDHURY; DESAI; AUDRETSCH, 2017). Embora o número de oportunidades aumente, barreiras sociais, educacionais e físicas ainda impedem que muitas pessoas tenham acesso as novas oportunidades (ACEMOGLU; ROBINSON, 2013).

A assistência social representa um dos meios de reduzir essas barreiras. Sendo no Brasil, o Sistema Único de Assistência Social (SUAS) responsável por regulamentar esse atendimento (COUTO *et al.*, 2014). Além do SUAS, o atendimento é realizado por organizações não governamentais e, em sua maioria, sem fins lucrativos. O Brasil, em 2012, contava com mais de 10,000 organizações, não governamentais, com algum fim de assistência social (IBGE, 2015).

Com o crescente número de informações, essas organizações passaram a necessitar da tecnologia da Informação (TI) para gerenciamento, comunicação, controle financeiro e estatístico de suas funções (LEE; CLERKIN, 2017). Um dos fatores contribuintes da expansão no uso de TI é a possibilidade de interação com o usuário final através da Internet (LEE; BLOUIN, 2015).

Embora o emprego de TI resulte em desempenho, muitas dessas organizações ainda operam sem o uso da tecnologia (TRIERVEILER; SELL; PACHECO, 2015). Os dois principais motivos são especificação e custo. Primeiramente, os serviços em grande parte são especializados em áreas da assistência social, tornando cada caso um produto diferente. Assim, por exemplo, um sistema capaz de atender múltiplos departamentos pode levar ao aumento de custo geral (desenvolvimento, manutenção) para um sistema com altas possibilidades de subuso (não usar todos os recursos do sistema). O segundo motivo é o custo para adquirir e manter as tecnologias, já que existem muitas dessas organizações que operam com custos mínimos, priorizando seu funcionamento e o bem estar dos envolvidos, ou, muitas vezes, com serviço voluntário sem formas de obter recursos.

Uma das associações que ainda trabalham sem um sistema de auxílio computacional é a Jovens Com Uma Missão (Jocum). A associação Jocum é uma entidade não governamental, sem fins lucrativos e com caráter cristão. Atuando por todo o mundo com a assistência de pessoas nas áreas de saúde, educação, assistência familiar e comunitária (JOCUM, 2014). A Jocum Ponta Grossa (Jocum PG) opera como uma entidade de Utilidade Pública desde 2001, e possui como objetivo o atendimento de crianças, adolescentes e famílias, respeitando a individualidade e singularidade de todos os envolvidos.

O serviço diário oferecido pela Jocum PG atende em média 100 alunos, entre 6 e 17 anos e seus familiares ou responsáveis. Além das crianças, são envolvidos no processo profissionais educacionais, psicólogos, pedagogos e outros. Os profissionais exigidos pelo SUAS e

o gestor municipal são mantidos através da parceria com a Fundação PROAMOR (JOCUM, 2014). As únicas outras formas de entrada de recursos são através da doação para instituição, ou através da doação das notas fiscais pelo programa Nota Paraná.

A fim de contribuir socialmente e oferecer a possibilidade de empregar um recurso de TI para a administração da Jocum PG, este trabalho objetiva a construção de um sistema gerenciador para as necessidades da organização. A construção busca auxiliar nas diversas atividades de gerenciamento das informações dos alunos e famílias, além dos profissionais envolvidos. Inicialmente, o propósito é oferecer uma ferramenta para os coordenadores, assistentes sociais e pedagogos, permitindo aos citados acessar as informações sobre as atividades referentes ao seu setor e aos projetos da organização.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é a criação de um sistema *web* para administração e controle dos envolvidos e, atividades empregadas pela Jocum de Ponta Grossa.

1.1.2 Objetivos Específicos

- Fazer a análise de requisitos junto à instituição Jocum.
- Analisar arquiteturas aplicáveis ao desenvolvimento *web*.
- Analisar ferramentas para implementação segundo a arquitetura definida.
- Desenvolver o servidor a partir das ferramentas selecionadas.
- Desenvolver um cliente que permita a manipulação das *views* de forma rápida.
- Construção do sistema como previsto.
- Avaliar o resultado.

1.2 JUSTIFICATIVA

A Jocum PG possui um fluxo médio de 100 alunos atendidos diariamente, somados aos familiares ou responsáveis e todos os seus funcionários, levando a um grande fluxo de dados. Estes dados, principalmente referente aos alunos, sofrem alterações e são manipulados para a geração de relatórios, estatísticas, acompanhamentos e outras funções.

Esta manipulação tornou-se ineficiente, uma vez que a grande quantidade de dados está

armazenada em arquivos de papel e existe a necessidade de diversos sistemas para apresentar de forma coerente os dados de seus programas sociais. Podem ser encontradas também outras instituições que ainda operam sem o uso de sistemas para a sua operação em busca de reduzir custos operacionais. Entretanto, esse uso torna-se ainda menos eficaz considerando o aumento dos dados armazenados, seja pelo aumento das atividades ou a extensão das mesmas por maiores períodos de tempo.

Este trabalho busca primeiramente oferecer o uso da TI para as administração das informações, atividades e envolvidos pela Jocum PG. A fim de possibilitar a centralização do trabalho dos envolvidos e permitir que o processo de acesso as informações torne-se mais eficiente que o modelo atual.

O sistema tem o objetivo de ajudar qualquer organização que careça de um programa gerenciador e que use os mesmos processos que a Jocum PG, podendo ser expandido dentro da organização Jocum ou externamente. O trabalho também visa a possibilidade de expansão dos recursos de forma livre, permitindo que novas opções possam ser facilmente implementadas. Desta forma, possibilitando que outras organizações ou mesmo a Jocum PG possa adequar o sistema às necessidades não abordadas neste projeto.

1.3 ORGANIZAÇÃO DO TRABALHO

O restante desse documento está organizado da seguinte forma: o Capítulo 2 descreve o levantamento de requisitos, análise e delimitação do escopo do projeto avaliando o tempo disponível. O Capítulo 3 aborda um estudo sobre as arquiteturas comuns para implementação de um projeto *web* e uma avaliação entre as mesmas. O Capítulo 4 aborda algumas das tecnologias mais usadas no mercado para *web*, considerando a avaliação das arquiteturas e as ferramentas disponíveis para as mesmas.

No Capítulo 5 são descritos os processos e funções desenvolvidas para atender a construção do sistema analisado. O Capítulo 6 descreve os resultados observados por no desenvolvimento do trabalho. O Capítulo final descreve as conclusões atingidas por este trabalho, as possibilidades futuras e limitações do desenvolvimento e análise

2 ANÁLISE DE PROJETO

Este capítulo objetiva a especificação do processo de levantamento de requisitos e da análise realizada para construção do sistema. Sendo esta uma das partes mais importantes no desenvolvimento, garantindo que o sistema desenvolvido corresponda ao seu propósito, sem desvios de funcionalidades ou desenvolvimento de funções não descritas (ALI; LAI, 2017).

Os requisitos de um sistema podem ser definidos como as condições/capacidades que devem atender o seu funcionamento. Segundo Machado (2011), esses requisitos podem ser entendidos como as regras de ação do sistema e qual a sua capacidade. Desta forma, uma das partes críticas na análise de um projeto é o levantamento de requisitos.

Para realização deste trabalho, foram realizados dois levantamentos de requisitos distintos. O primeiro, destinado a estabelecer um mapeamento das funcionalidades desejadas do sistema, opções de expansão e formas de uso do sistema por todos os usuários. Através da análise inicial, então foi estabelecida uma verificação das funcionalidades principais, assim como as funções mínimas para a implantação do sistema, dentro do fator mais crítico neste trabalho (tempo). A forma de desenvolvimento também deve fornecer uma base sólida para que as funções adicionais sejam facilmente implantadas posteriormente, sem a perda de eficiência, escalabilidade ou necessidade de alteração na estrutura do sistema.

2.1 LEVANTAMENTO DE REQUISITOS

O levantamento de requisitos iniciais foi realizado através de entrevista. Como os principais responsáveis e maiores envolvidos no processo de gerenciamento dos dados, o coordenador do Jocum PG e a assistente social, foram os primeiros entrevistados, como solicitantes do projeto e principais atendidos pelo mesmo.

Uma característica importante do levantamento de requisitos, é a distinção entre os requisitos funcionais e não funcionais. Os requisitos funcionais são as descrições das funcionalidades ou operações,as quais os usuários necessitam e definem a utilidade e escopo de um sistema (MACHADO, 2011). Essas operações podem ser realizadas através de um comando do usuário ou devido a outros eventos no sistema.

Os requisitos não funcionais, não são menos importantes, uma vez que definem as características de funcionamento estrutural e alguns dos padrões de qualidade que devem ser atendidos pelo sistema, como desempenho/eficiência, portabilidade, escalabilidade, segurança e outras (MACHADO, 2011).

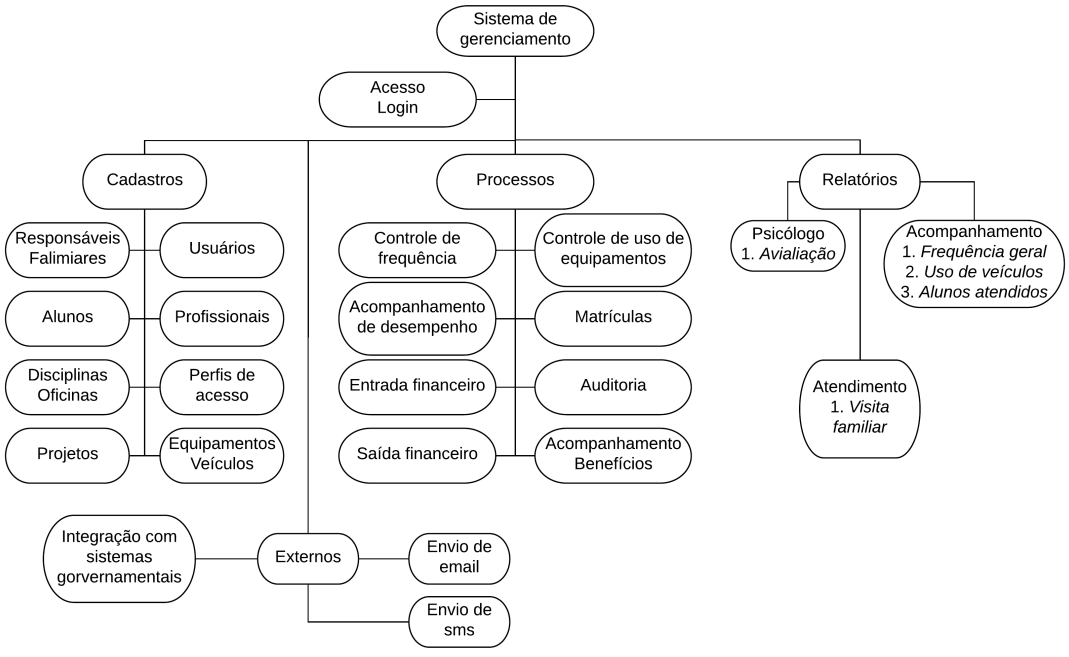
A entrevista inicial, dedicou-se principalmente ao levantamento dos requisitos funcionais, uma vez que os membros da equipe solicitante apresentaram pouco conhecimento para discussão dos requisitos não funcionais, neste primeiro momento. A entrevista foi conduzida

através de um *brainstorm* para analisar as necessidades dos envolvidos, bem como a importância de cada funcionalidade. Após esse levantamento, foi realizado um questionário, para expansão e aprofundamento das ideias previamente levantadas e cobertura de requisitos não previstos.

O questionário foi aplicado também para verificação de requisitos não funcionais, os quais os usuários não estão familiarizados. O estabelecimento tardio destes requisitos pode levar a problemas no decorrer do projeto, considerando principalmente a relação de esforço ou mesmo viabilidade do projeto (ABDUKALYKOV *et al.*, 2011). Desta forma, o questionário também buscou estabelecer os requisitos de funcionamento e as opções disponíveis para o desenvolvimento do sistema, considerando suas funcionalidades e, os mecanismos de segurança, de desempenho e de *hardware* necessários.

A partir da entrevista completa, foram mapeadas as funcionalidades do sistema. A Figura 1 apresenta o resultado do mapeamento de todos os requisitos solicitados ou desejáveis, sob uma análise estrutural do sistema. Na Figura 1 o sistema é divido nos formatos de cadastros, processos envolvidos e relatórios gerados pelo sistema, sendo enumeradas apenas as informações levantadas na entrevista.

Figura 1 – Diagrama dos requisitos funcionais.



Fonte: Autoria Própria.

O Quadro 1, representa o levantamento dos requisitos não funcionais levantados através da entrevista. É possível verificar através do quadro, que categorias como tempo de resposta, tecnologia usada e outras não foram definidas, sendo parte deste projeto a exploração de opções de arquiteturas e tecnologias disponíveis para o desenvolvimento.

Quadro 1 – Requisitos não funcionais.

Identificador	Descrição
RNF001	A base de dados deve ser protegida para acesso apenas de usuários autorizados.
RNF002	Utilização do sistema em rede local ou web (Internet).
RNF003	Uso de Design responsivo nas interfaces gráficas.
RNF004	Compatibilidade com sistemas operacionais Windows e Linux.
RNF005	Divisão arquitetural do sistema em camadas para desacoplamento.
RNF006	Alta capacidade de armazenamento de arquivos.

Fonte – Autoria Própria

2.1.1 Definição do escopo

Segundo Kerzner e Kerzner (2017), existem diversas causas para a falha de um projeto. Entre elas deve ser citado a falha em definir os recursos disponíveis, como capital, tempo e conhecimento. Desta forma, uma das preocupações deste projeto é o compromisso em entregar um sistema computacional com base sólida para o uso inicial, já com as funcionalidades mínimas instaladas, e as funções desejadas a serem adicionadas posteriormente.

Desta forma, após o mapeamento inicial das funcionalidades, foi realizada uma nova entrevista com os principais interessados, para definição das limitações do projeto a serem entregues ao final do trabalho. A partir desta, foi estabelecido que o projeto será construído em ciclos.

Cada ciclo definido através do nível de importância das funções mapeadas e o tempo previsto para que cada ciclo fosse finalizado. Após a escolha do formato de entrega e definição das prioridades de cada requisito funcional, foi possível definir qual modelo de desenvolvimento seria adequado a este trabalho.

2.1.2 Modelo Iterativo e Incremental

O modelo Iterativo e Incremental é o modelo que divide o desenvolvimento do produto em ciclos. A cada novo ciclo, são realizadas novamente as fases de análise, projeto, desenvolvimento e testes, sendo referente apenas a uma parte do projeto final (BEZERRA, 2017).

Considerando o mapeamento dos requisitos, foram separados os grupos iniciais, considerando prioridades das funcionalidades, tempo de desenvolvimento, tempo para testes de desenvolvimento, e a possibilidade de uso do sistema em expansão, sem comprometimento das partes desenvolvidas em ciclos anteriores. Outra característica importante é a possibilidade de avaliar o comportamento dos requisitos não funcionais.

As principais vantagens desse modelo são a possibilidade de alterações durante o desenvolvimento, redução de riscos, aumento da velocidade de desenvolvimento, entre outros. Entretanto, há desvantagens a serem consideradas antes da escolha do modelo, como o risco do produto final não atender todas as exigências, problemas de arquitetura e o custo/tempo do projeto ser maior que o previsto (BEZERRA, 2017).

O fator financeiro apresentou um risco menor, uma vez que o custo de produção foi virtualmente zero, e serão usados os recursos já disponíveis pela organização para a implantação do sistema. Para este trabalho, o principal risco foi o tempo, uma vez que alguns requisitos essenciais podem ter sido ignorados devido a inúmeras situações, como falta de conhecimento das regras de negócio, por parte do desenvolvimento ou regras de tecnologia por parte dos solicitantes.

A fim de reduzir esse risco ficou estabelecido um tempo maior para os dois primeiros ciclos do projeto, onde seria construída a base do sistema, as funcionalidades de acesso e cadastros gerais. Outra medida para a redução dos riscos do projeto foi a separação dos ciclos a serem entregues no tempo e a finalização dos complementares.

Um tempo considerado normal para a execução de um ciclo é de duas semanas, considerando o número de membros da equipe por itens a serem desenvolvidos (BEZERRA, 2017). Desta forma, limita-se o desenvolvimento em partes menores que possam ser executadas dentro do período estabelecido. Neste trabalho, foram definidos os tempos de execução com período de três semanas, entre tempo de desenvolvimento, testes e escrita. O Quadro 2 apresenta uma relação dos ciclos, como as funcionalidades e tempo previstos para sua execução.

Quadro 2 – Ciclos de desenvolvimento.

Identificador	Descrição
C-01	Definição da arquitetura, tecnologias usadas, modelagem do banco de dados e construção da estrutura básica da arquitetura escolhida.
C-02	Implementação da base para as tecnologias responsivas e modelagem do acesso, cadastro de perfis e usuários.
C-03	Modelagem do controle de matrículas, profissionais, alunos, responsáveis e disciplinas atendidas.
C-04	Modelagem do cadastro de equipamentos, controle de frequência, acompanhamento estatístico dos alunos e controle de uso dos equipamentos.
C-05	Modelagem do processo de auditoria sobre a manutenção dos cadastros e processos. Modelagem do cadastro de projetos e , implementação para armazenamento de arquivos em formatos diversos.
C-06	Implementação da base para geração de relatórios e fichas de acompanhamento.
C-07	Modelagem do módulo de acompanhamento financeiro.
C-08	Implementação de recursos externos, para envio de email, SMS e comunicação com outros sistemas.

Fonte – Autoria Própria

Observa-se que a lista apresenta a definição de todos os ciclos previstos no projeto, mas que apenas os três primeiros ciclos foram implantados e analisados dentro deste projeto. O motivo, é que este trabalho destinou-se a resolver o problema inicial encontrado pela Jocum PG, armazenamento dos dados referentes ao seu serviço, que é o atendimento dos alunos, seus cursos, e os profissionais que utilizaram o sistema. Outra limitação foi o tempo, que inviabilizou a produção do sistema com todos as funcionalidades desejadas dentro do período de execução deste trabalho.

Os requisitos específicos referentes a cada ciclo foram obtidos através de novas entrevistas e análise dos documentos armazenados pela Jocum PG. As entrevistas buscam entender o processo e regra de negócios do sistema, enquanto a análise dos documentos visa a obtenção do modelo de dados armazenados, além dos tipos dos dados e suas interligações.

3 ARQUITETURAS

Construir sistemas que apresentem capacidade de escalabilidade e desempenho, e um dos desafios de um projeto voltado à *web*. Essas dificuldades se apresentam, principalmente, devido a grande quantidade de recursos, dados e usuários atendidos por um sistema em expansão. De acordo com Pressman e Maxim (2016), a arquitetura de um sistema deve ser definida através da análise da organização a qual um sistema deverá compor e sua estrutura geral. Como este trabalho visa essas duas características, é fundamental estudar algumas das arquiteturas disponíveis, voltadas a resolução deste desafio.

Neste capítulo são abordados alguns dos conceitos de arquiteturas destinadas a *web*, as quais oferecem eficiência e permitem a escalabilidade. As características que as compõem, e os meios de comunicação e o uso de *threads*.

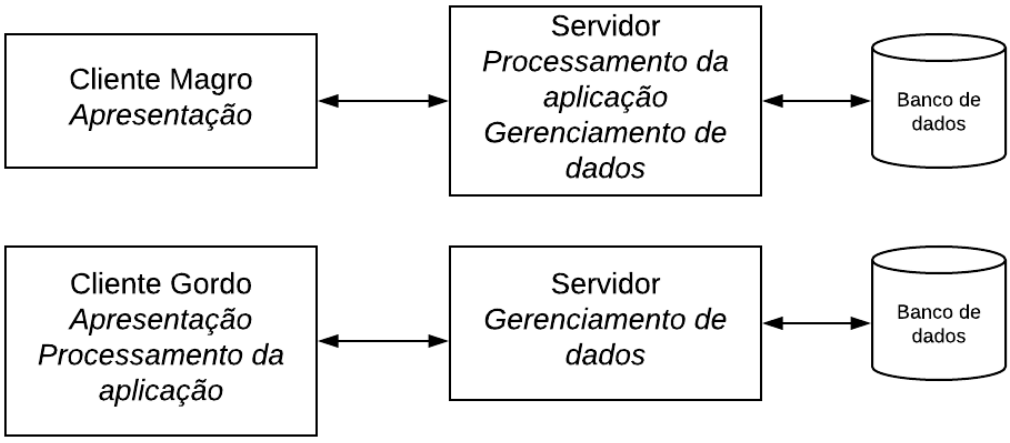
3.1 CLIENTE E SERVIDOR

A partir da necessidade do processamento descentralizado, através da rede (principalmente Internet) criou-se a arquitetura conhecida como cliente-servidor, ou cliente/servidor de duas camadas. Nesta arquitetura, são distribuídos o processamento entre as duas camadas, sendo cada uma delas responsável por executar um processamento específico, e tornando cada camada um serviço especializado em uma funcionalidade (GOODYEAR, 2017).

Em sua forma mais comum, a camada de cliente fornece os serviços de interação com o usuário, através da inserção, alteração, exclusão, manipulação ou simplesmente visualização dos dados. A camada servidor, tem como objetivo o processamento dessas entradas e o gerenciamento do armazenamento dos dados e acesso a serviços adicionais. Esses, podem ser autenticação de acesso, geração de relatórios e as estatísticas, ou simplesmente do salvamento dos dados para futuras consultas (LAUDON; LAUDON, 2016).

Apesar de serem visões distintas sobre a organização das funcionalidades, existe a possibilidade de visões alternativas sobre como as funções devem ser divididas entre as duas camadas. Sommerville (2011) aponta, por exemplo, que a arquitetura pode ser construída a partir de um "cliente-magro"ou "cliente-gordo", demonstrado através da Figura 2.

Figura 2 – Arquiteturas cliente-servidor segundo Sommerville (2011).



Fonte: Modificado de (SOMMERVILLE, 2011).

No caso do "cliente-magro", a camada de cliente destina-se apenas a apresentação dos dados, e o servidor ocupa-se do processamento da aplicação, processamento dos dados e gerenciamento do acesso ao banco de dados. Um exemplo de funcionamento seria o processamento os recursos no servidor, e apenas a renderização (apresentação em tela) no cliente. Este processamento pode levar ao aumento do tráfego entre o cliente e servidor, uma vez que muitas informações terão que ser repassadas para o cliente. Também acarreta no aumento do processamento por parte do servidor. Em contrapartida, o "cliente-magro"exige menos recursos de memória e do cliente (SOMMERVILLE, 2011).

O "cliente-gordo", diferencia-se por realizar o processamento da aplicação no lado do cliente. Desta forma, recursos tais como o processamento da aplicação, validações e outras funcionalidades podem ser realizadas do lado do cliente. Este tipo de processamento pode dividir o "custo"de operação com o cliente, uma vez que o processamento inicial não ocorre diretamente no servidor.

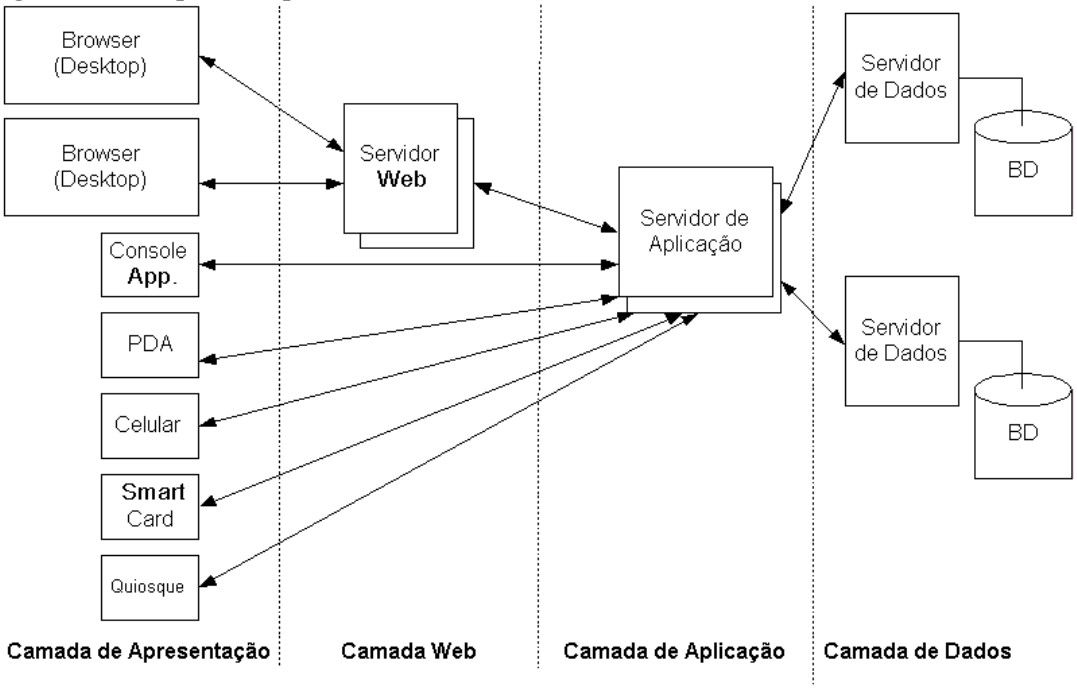
Assim, é importante considerar, ao usar esta arquitetura quais os recursos disponíveis para o servidor e quais os recursos disponíveis para o público alvo, de forma que estes possam usar o sistema sem o gasto de tempo excessivo com processamento dos dados. Coulouris, Dollimore, Kindberg, e Blair (2013), em seu livro, afirmaram que a arquitetura cliente-servidor está entre as mais importantes na área de sistemas distribuídos (historicamente), e é de uso comum por todo o mundo.

3.2 MULTICAMADAS

Segundo Sommerville (2011), ao estender o modelo cliente-servidor para multicamadas, pode-se alcançar maior escalabilidade, uma vez que servidores extras possam ser adicionados e que o processamento seja balanceado entre os mesmos.

Este cenário, também conhecido como N-camadas, é mais utilizado por grandes empresas, ou empresas que oferecem sistemas com grande número de usuários. Através do uso das N-camadas, como na Figura 3, pode-se dedicar o processamento dos dados, separadamente, de acordo com o serviço acionado, reduzindo tempo de espera, uma vez que há menos disputa por recursos comuns de *hardware*, tais como acesso a rede, de memória, o processamento.

Figura 3 – Exemplo de arquiteturas n-camadas



Fonte: (SAUVE, 2016).

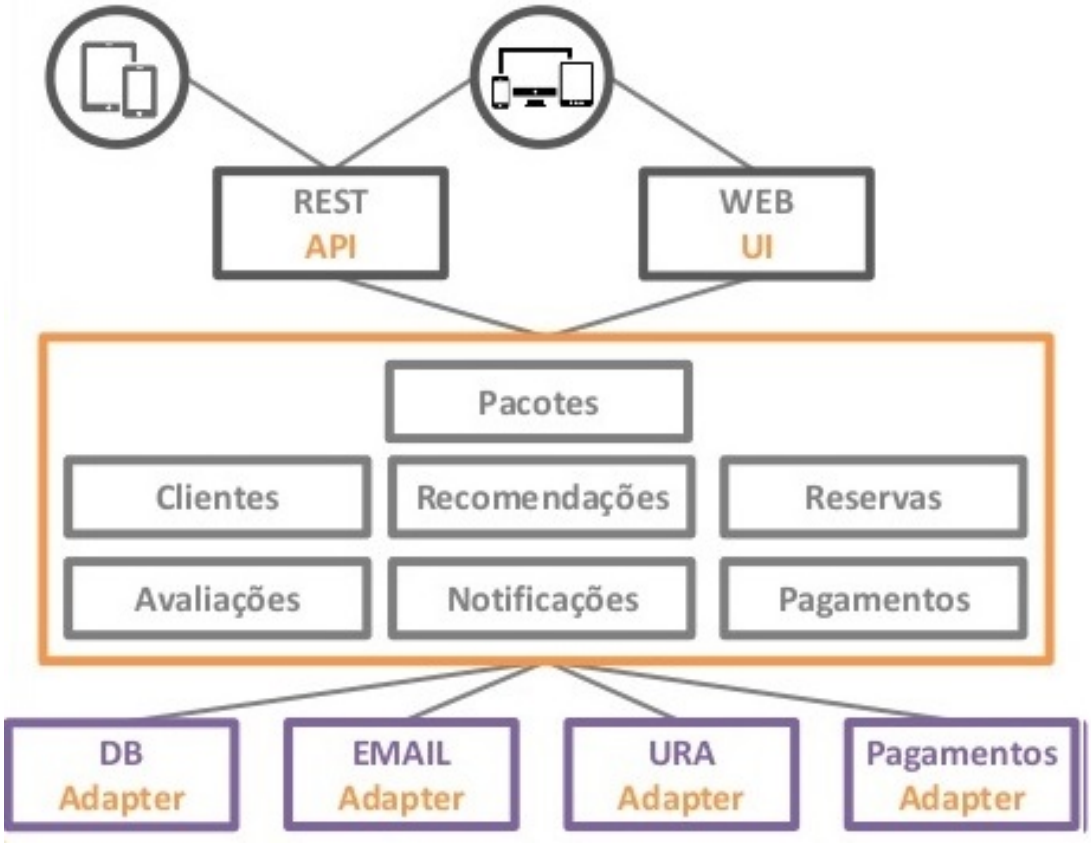
Desta forma, é possível também aumentar o processamento para as camadas que mais são usadas, e reduzir o processamento das camadas com menor uso.

3.3 MICROSERVIÇOS

De acordo com Savchenko, Radchenko e Taipale (2015), os microserviços são um padrão de desenvolvimento de aplicativos para nuvem, que usa a divisão de suas funcionalidades em serviços menores e específicos. Os microserviços também podem ser vistos como meta processos, que são independentes entre si, podendo existir duplicatas e estarem em estados suspensos ou ativos sem o comprometimento dos demais, comunicando-se com os demais através da troca de mensagens (NEWMAN, 2015).

Aplicações *web*, tradicionalmente, utilizam a chamada arquitetura monolítica, onde a grande maioria de seus recursos são executados em um único processo. Nos casos onde há necessidade de paralelismo dos recursos, balanceamento de carga e uso de sistemas atrelados é menor, a arquitetura apresenta vantagem, pois possuem uma estrutura mais simples, quando comparado aos microserviços. Um exemplo da estrutura monolítica pode ser observada na Figura 4.

Figura 4 – Exemplo de sistema de viagens monolítico

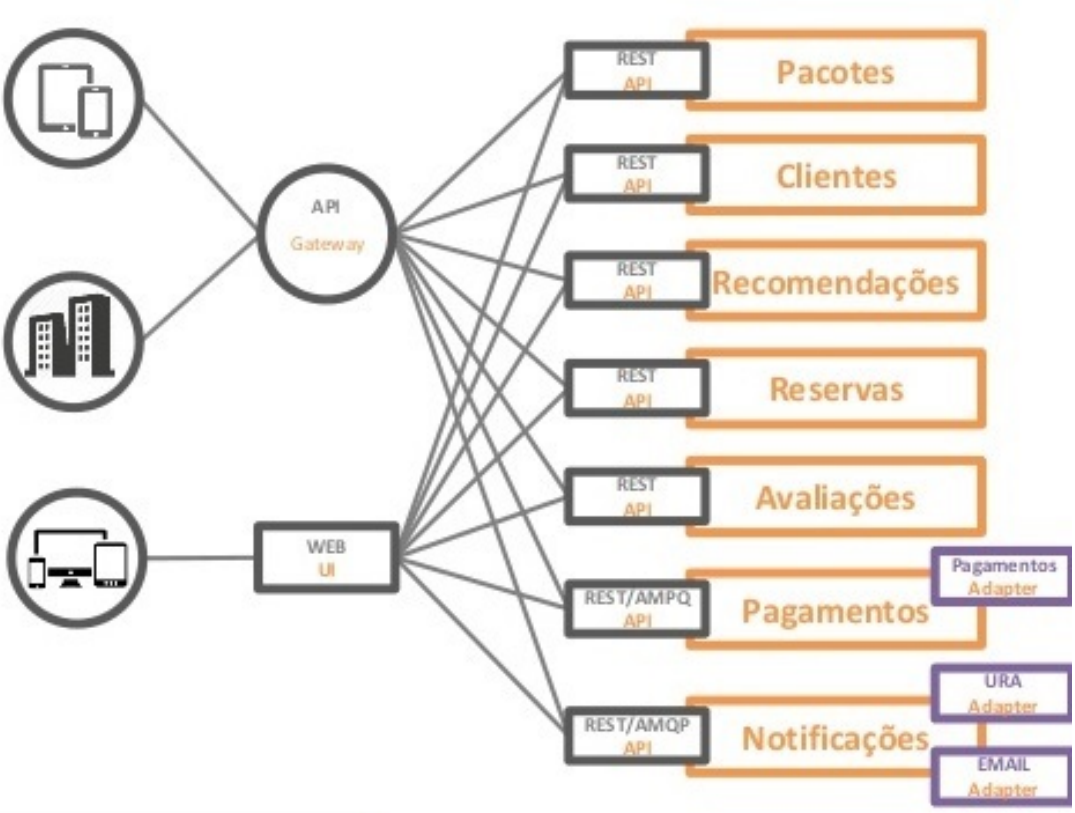


Fonte: Modificado de (ROSATO, 2015).

No entanto, conforme a demanda e o consumo dos recursos computacionais é aumentado, a complexidade para a manutenção destes sistemas também aumenta (STUBBS; MOREIRA; DOOLEY, 2015). Alterações de componentes podem facilmente afetar o funcionamento de outras partes, comprometendo a integridade da aplicação, reduzindo sua eficiência e impactando no uso e evolução do sistema.

A Figura 5 demonstra como pode ser arranjado o funcionamento de uma plataforma de viagem usando micro-serviços. Observa-se que diferentemente da estrutura monolítica (Figura 4), os serviços são disponibilizados separadamente.

Figura 5 – Exemplo de sistema de viagens usando microservices



Fonte: Modificado de (ROSATO, 2015).

Entre as vantagens de sua utilização, Newman (2015) cita:

- Heterogêneos: Como os serviços são independentes, é possível usar diferentes tecnologias, adequando o serviço com a tecnologia usada, sem comprometer o funcionamento do sistema em seu todo. Permite o uso da tecnologia mais adequada ao cenário pretendido.
- Resiliência ou isolamento: Neste modelo é possível isolar componentes com erros, sem a necessidade de interromper o funcionamento do sistema, que pode ainda continuar com funcionamento completo, caso existam réplicas do serviço que não apresentem o mesmo erro.
- Manutenção ágil: Os serviços podem receber manutenção de forma individual, facilitando testes, implementação e implantação do serviço, sem comprometer totalmente o sistema.
- Custo de expansão: Há uma redução considerável no custo para expansão dos seus recursos, uma vez que pode-se buscar pela alternativa mais adequada e de melhor custo para uma função específica, não sendo limitados a atender todas as necessidades de todas as funcionalidades do sistema.

3.4 EFICIÊNCIA E ESCALABILIDADE

Segundo Liu (2011), a eficiência de um sistema, é a medida de quão veloz um sistema é capaz de executar tarefas computacionais, enquanto escalabilidade representa uma estimativa do desempenho com o aumento progressivo de carga. A eficiência pode ser medida de diversas formas. Por exemplo, o processamento de transações, na qual é medido o tempo entre a requisição de um usuário até sua resposta. Outro o processamento em lote, onde é avaliado o número de transações que um sistema pode resolver dado um limite de tempo (LIU, 2011). Através desta visão, é possível perceber se um sistema pode ser eficiente, mas não demonstrar se pode ser escalável.

Essa visão, também demonstra que é inviável a escalabilidade para um sistema ineficiente, uma vez, que levaria a maiores problemas de eficiência. Todavia, se um sistema eficiente pode ser melhorado através do aumento com uso de *hardware* adicional, este sistema é escalável.

Uma das possíveis verificações de comportamento de um sistema com o aumento da carga, é a investigação, monitoramento do desempenho dos pontos mais utilizados do sistema (CHIEU; MOHINDRA; KARVE, 2011). Estão entre os indicadores mais usados para sistemas *web*, o número de usuários simultâneos (concorrentes), número de conexões paralelas, número de requisições em um determinado tempo e quanto tempo estas requisições consomem.

Dada a possibilidade de escalabilidade, a mesma pode ocorrer de duas formas, horizontal e vertical. A escalabilidade horizontal é o aumento do número de máquinas para suportar o crescimento do sistema, enquanto a vertical representa o aumento de hardware em um nó, com o aumento do número de processadores, memória de acesso randômico (RAM), uso de *solid-state drive* (SSD) e outras tecnologias (ANWAR, 2018).

Embora as memórias, capacidade de disco e processadores aumente gradualmente, ainda existe uma grande limitação para a escalabilidade vertical em comparação a horizontal.

3.5 COMUNICAÇÃO SÍNCRONA OU BLOQUEANTE

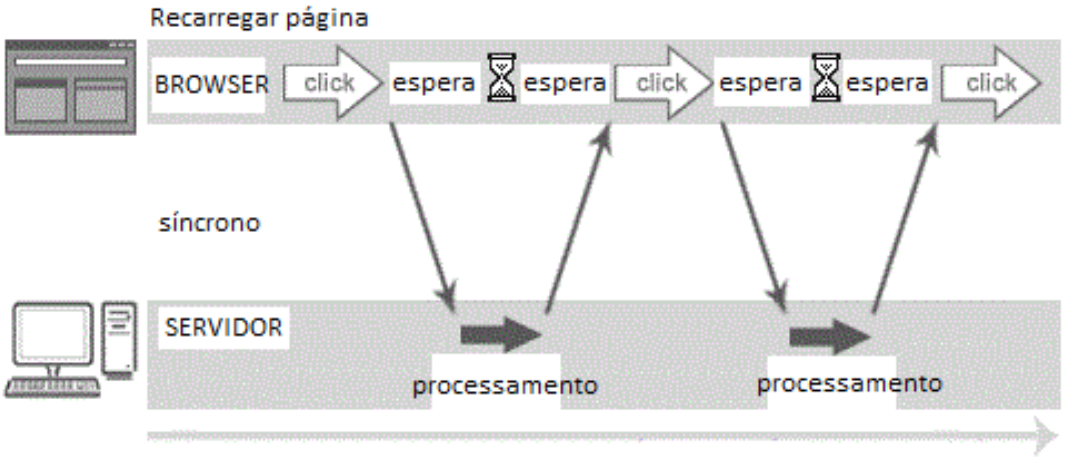
Uma parte importante da escolha das arquiteturas é o formato de comunicação entre os componentes do sistema. A programação tradicional, executa a comunicação de entradas/saídas (E/S), com a execução de funções sequencialmente, de forma bloqueante. Isso implica em informar que uma função deve ser encerrada antes que outra seja processada, isolando usuários, e de forma que as operações não concorram entre si.

Com a comunicação bloqueante, é mantida uma fila de funções com grande ociosidade, enquanto outra operação é executada. Ao enviar, por exemplo, um e-mail, consultar dados no banco ou fazer gravações de dados em disco, o sistema permanece aguardando a execução de tarefas futuras, bloqueando atividades que poderiam ser executadas em paralelo (TEIXEIRA,

2012).

A Figura 6 mostra um exemplo do carregamento de uma página web com a comunicação síncrona. Pode-se observar que entre cada *click* existe um tempo de aguardo, no qual não são processadas outras informações até o recebimento de uma resposta.

Figura 6 – Exemplo de comunicação síncrona.



Fonte: Modificado de JAVATPOINT 2015

Embora o modelo apresente grandes períodos de ociosidade, este tipo de comunicação tende a garantir que ocorram menores quantidades de transmissão de informações desatualizadas, uma vez que todas as funções seguintes terão disponíveis o que suas predecessoras realizaram.

Ainda segundo Teixeira (2012), existem alternativas para o modelo bloqueante, como o uso de *multi-threads*, que consiste no partilhamento de recursos computacionais entre diversos trechos de um mesmo processo. Cada *thread* consiste em um pequeno subsistema, que divide um processo em duas ou mais tarefas. Assim, cada E/S pode ser tratada por uma *thread* diferente, podendo ser executadas paralelamente, reduzindo o tempo para execução da comunicação. Além do partilhamento de memória, os sistemas podem ainda utilizar dos hardwares multiprocessadores para executar paralelamente os processos em núcleos diferentes da Unidade Central de Processamento (CPU).

Uma das desvantagens do uso de *multi-threads* é a perda de controle sobre a ordem de execução das *threads* e seu acesso à memória. Em aplicações que possuem uma dependência do controle dos estados da comunicação e ordem em que é executada, o uso das *threads* pode ocasionar em erros de *deadlock*, onde um processo A impede o outro B, enquanto B impede A de continuar, ou outros problemas (TEIXEIRA, 2012).

Pereira (2014) enfatiza ainda, sobre a comunicação bloqueante, o aparecimento de gargalos no desempenho do sistema, à medida que o número de acessos aumenta, sendo necessários *upgrades* dos *hardwares* dos servidores ou mesmo da tecnologia usada, a fim de reduzir o tempo ocioso para as tarefas síncronas. Teixeira (2012), também alerta que este tipo de comunicação

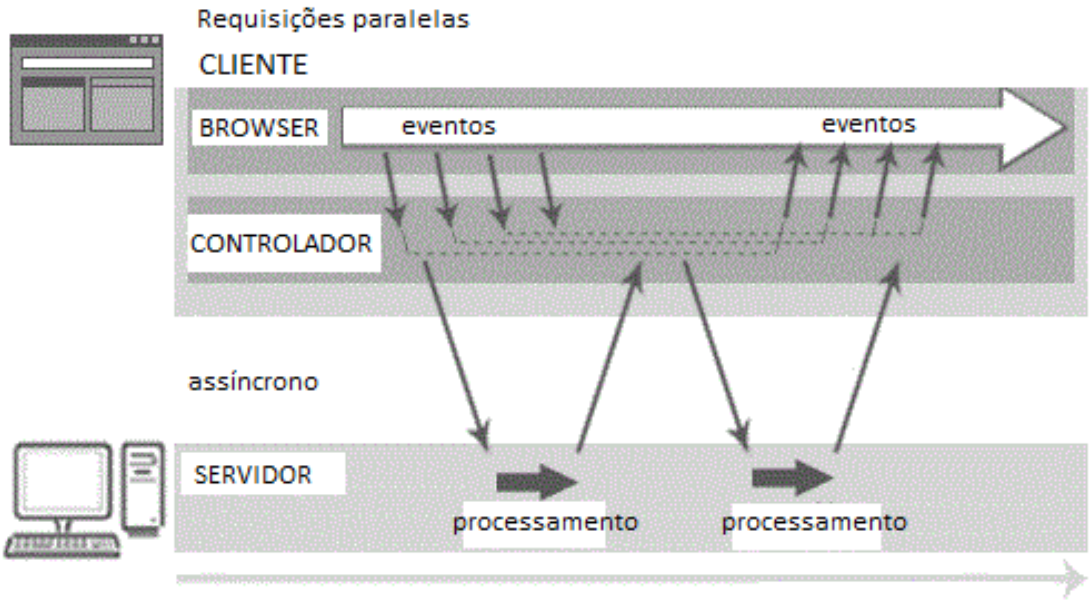
tende a oferecer pouca escalabilidade, quando não usada horizontalmente, principalmente em ambiente de rede.

3.6 COMUNICAÇÃO ASSÍNCRONA OU NÃO-BLOQUEANTE

Entre as estruturas mais comuns para arquiteturas não-bloqueantes estão os modelos de programação orientados a eventos. Esta programação consiste no controle do fluxo de eventos que ocorrem durante a execução. Os eventos são manipulados através de controladores. Estes controladores, conhecidos como *callbacks* são acionados a cada acontecimento significante do sistema, como retorno de um acesso ao banco ou acionamento de um botão.

O desenvolvedor define quais funções serão executadas a partir de ocorrências relevantes no sistema. Neste processo, as operações de E/S ocorrem de forma paralela, e cada *callback* é chamado após o tratamento que o ocasionou (TEIXEIRA, 2012). Assim, a troca de mensagens não exige que seja aguardado um retorno para a mensagem, a fim de continuar a comunicação, como demonstrado na Figura 7. Invocações não-bloqueantes são excelentes opções para executar rapidamente rotinas, sem interromper serviços, através da chamada de eventos (PEREIRA, 2014).

Figura 7 – Exemplo de comunicação assíncrona.

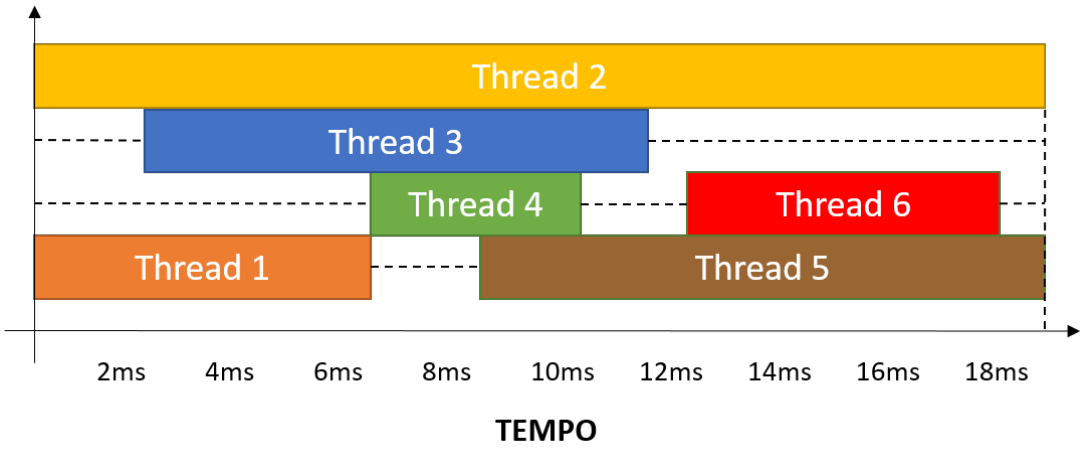


Fonte: Modificado de (JAVATPOINT, 2015)

3.7 *THREADS*

Threads são abstrações de atividades, que podem ser executadas em paralelo (exemplo Figura 8), e são controladas através do sistema operacional (S.O.). A execução de múltiplas delas resulta na execução de um processo, a qual tem sua criação ou destruição sob controle do S.O. (COULOURIS *et al.*, 2013). A opção de executar múltiplas *threads* está disponível na maioria dos S.O. modernos, embora existam autores, como Cordeiro (2006), que defendem o uso apenas para situações que exigam simultaneidade.

Figura 8 – Exemplo de *threads* paralelas.



Fonte: Modificado de (SCHIECK, 2017)

Diferentemente da arquitetura orientada a eventos, existem algumas barreiras no uso de *multi-threads*. Entre elas, a depêndencia de módulos, gerenciamento da memória, e possível ocorrência de *locks* ou *deadlocks*, dificultam o uso em sistemas que exigem escalabilidade.

Por outro lado, Cordeiro (2006) ressalta que *multi-threads* pode representar a melhor opção para casos onde existam operações concorrentes. Embora, a utilização de orientação de eventos ainda represente a opção mais eficiente no processamento simultâneo de requisições.

3.8 CONCLUSÕES

Pesquisas são realizadas a décadas, sobre o uso de *threads* e arquiteturas baseadas em eventos, sendo em muitos casos, a decisão de qual arquitetura escolher, uma responsabilidade do analista ou da equipe de projetos verificar a solução que melhor se encaixe aos cenários do sistema e as funções executadas em paralelo (DABEK *et al.*, 2002).

Uma característica importante a ser observada para os projetos web é o crescimento do número de servidores web e de computadores, *smartphones* e outros dispositivos capazes de acessar a rede Internet (COULOURIS *et al.*, 2013). Para comportar essa expansão, o sistema

deve ser capaz de atender à demanda e, também, que sejam adicionados novos recursos em qualquer ponto da vida útil do sistema (CHIEU; MOHINDRA; KARVE, 2011).

Sistemas que não permitem essa expansão ou são ineficientes estão fadados a terem uma curta vida útil, causando um desperdício de recursos por parte do desenvolvimento e dos usuários. Nenhum dos métodos oferece uma solução perfeita, sendo uma tendência optar pelo arquitetura de eventos, uma vez que as possibilidades futuras são mais amplas.

4 TECNOLOGIAS

Este capítulo dedica-se a apresentar tecnologias que foram aplicadas a este trabalho. Inicialmente, analisando algumas das tecnologias que atendem o paradigma de orientação a eventos. São analisadas também as dependências, como bibliotecas e outros recursos, mais usadas por elas. Em seguida, é abordado uma comparação entre opções para o lado cliente, que deve ser responsivo, segundo requisito. Após, apresenta-se uma comparação entre dois dos bancos de dados mais usados comercialmente e suas características para definir a escolha mais adequada.

4.1 SERVIDOR

A partir desta arquitetura, foi apresentado também uma visão sobre *frameworks* disponíveis e que podem ser implantados usando a arquitetura escolhida. Entre os inúmeros *frameworks*, observa-se que mais de 50% são desenvolvidas em JavaScript ou PHP (HOTFRAMEWORKS, 2018). Desta forma, foram avaliados características dos *frameworks* Node (javascript) e Laravel (PHP) para o servidor.

4.1.1 Node

A plataforma Node.js é definida por sua agilidade no desenvolvimento e sua escalabilidade para aplicações em rede. Foi construído para operar de forma leve e eficiente, de forma orientada a eventos, adequando-se a aplicações de tempo real (TILKOV; VINOSKI, 2010). Além destas características, o Node.js é uma ferramenta de código aberto e gratuito, que possui compatibilidade com diversos sistemas operacionais.

Logo, pode-se afirmar que o Node.js é uma maneira de construir sistemas que possuirão concorrência, recursos escaláveis e são puramente orientados a eventos que não bloqueiam a infraestrutura a qual utilizam (TILKOV; VINOSKI, 2010). Para realizar isso, o Node.js foi construído com o interpretador V8 de Javascript do navegador Chrome, do Google. Sendo desenvolvido por desenvolvedores da própria empresa e contando com uma enorme comunidade. A execução na engine V8 foi ajustada para trabalhar melhor em contextos além do navegador, principalmente, com fornecimento de APIs otimizadas para casos de uso específicos.

Uma das principais características do Node.js é o uso da linguagem JavaScript como recurso de desenvolvimento, que é uma das linguagens mais populares do mundo. Além de amplamente usada, a linguagem é pode ser executada quase em qualquer S.O (CANTELON *et al.*, 2017). Outra vantagem da linguagem é sua constante evolução e aprimoramento, uma vez que empresas como Google, Mozilla, Apple, entre outras, usam a linguagem em seus navegadores.

Um exemplo desta evolução, é o emulador JSLinux (BELLARD, 2017), capaz de carregar o kernel do Linux, possibilitando o uso do terminal de comando, compilação de programas em C e outras funções, completamente através do navegador.

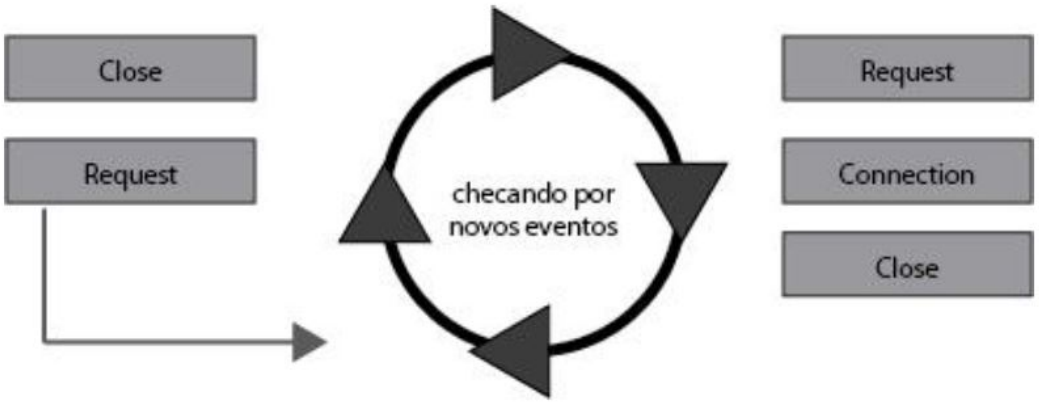
Segundo Cantelon *et al* 2017 exitem benefícios adicionais que podem ser mencionados devido ao uso de JavaScript, como:

- Uso de uma mesma linguagem para servidor e cliente, favorecendo reuso de código para algumas funções.
- Troca de mensagens no formato JavaScript Object Notation (JSON), deixando a comunicação mais organizada.
- Possibilidade de uso como bancos de dados NoSQL, uma vez que alguns desses bancos armazenam diretamente mensagens JSON.
- O interpretador é atualizado junto ao padrão ECMAScript, garantindo que o Node.js seja capaz de acompanhar a evolução da linguagem, oferecendo aos desenvolvedores melhores opções.
- O interpretador é considerado rápido, com seu desempenho definido através dos princípios de acesso rápido a propriedade, geração de código dinâmico e coleta de lixo eficiente.

O Node.js é construído a partir de duas divisões de bibliotecas. A primeira, responsável pelo laço de eventos de E/S, de forma rápida para redes ou sistemas de arquivos, como exemplo a libuv. A segunda divisão corresponde a biblioteca Protocolo de Transferência de Hipertexto Seguro (HTTP), na qual estão definidos os métodos de manipulação do seu protocolo (WILSON, 2018).

De acordo com Teixeira (2012), a disponibilidade de funções de primeira classe e recursos como *closures*, são alguns dos fatores que tornam o Node uma das ferramentas mais adequadas a orientação a eventos. A forma de processamento dos eventos, conforme explicação de Cantelon *et al* 2017, é chamado de Event-Loop (EL). O EL, apresentado na Figura 9, é um mecanismo interno, que utiliza de bibliotecas para manipular e gerenciar os eventos, com funcionamento ininterrupto, onde cada interação verifica os estados dos eventos.

Figura 9 – Event-Loop no Node.js.



Fonte: (PEREIRA, 2014)

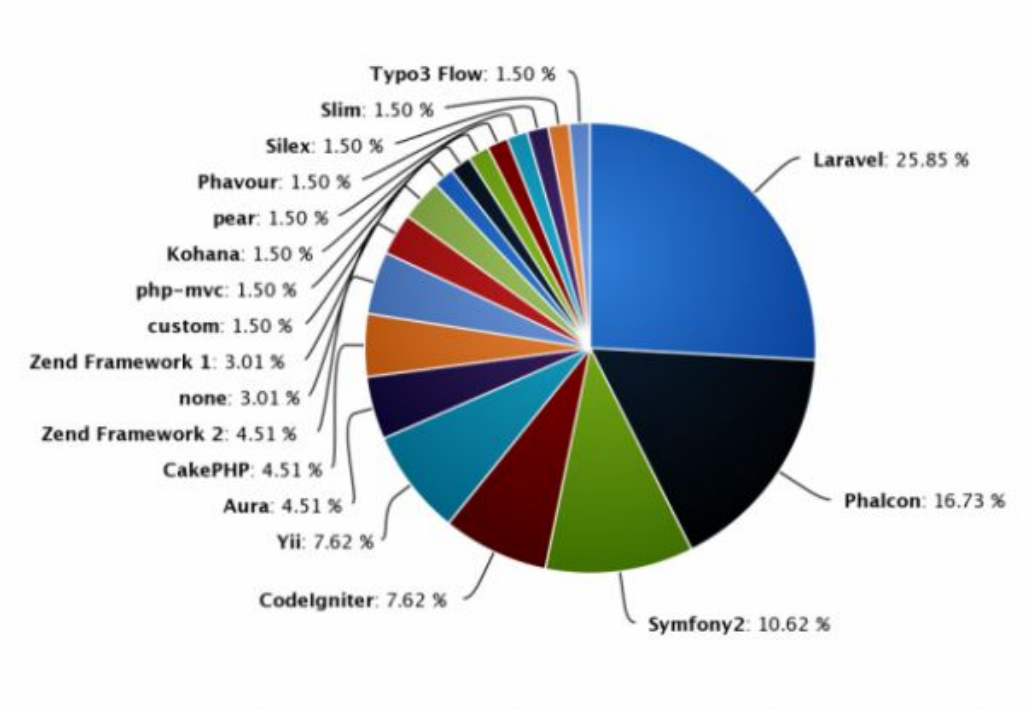
4.1.2 Laravel

O Laravel é um *framework* de desenvolvimento PHP, livre e de código aberto, cujo principal objetivo é permitir trabalhar de forma estruturada e rápida (DOUGLAS; MARABESI, 2017). O sistema é construído para utillizar o padrão MVC (*Model View Controller*), dividindo o desenvolvimento entre camadas (modelo, visão e controlador).

Uma de suas características é a possibilidade de desenvolver *templates* através de uma *Engine*, conhecida como Blade. Essa *Engine* possui uma variedade de ferramentas e opções, que buscam facilitar a criação das interfaces gráficas, com boa apresentação e de forma funcional. O uso do Blade também busca a redução de códigos duplicados (DOUGLAS; MARABESI, 2017). Outra característica do framework citado é a busca por oferecer uma solução simples e de fácil aprendizado.

Além de sua praticidade, o Laravel é amplamente usado no desenvolvimento de sistemas para o mercado, como demonstrado no estudo de (SKVORC, 2013). Na pesquisa, o Laravel apresentou uma margem considerável, na popularidade, quando comparado com outros *frameworks* para desenvolvimento com o PHP. O resultado da pesquisa, pode ser observado na Figura 10, que apresenta o gráfico de popularidade dos *frameworks* PHP.

Figura 10 – Popularidade dos *frameworks* PHP em 2013.



Fonte: Modificado de SKVORC (2013)

Douglas e Marabesi (2017) citam que este *framework* permite a execução de ações, como configuração de ambiente, ações com a própria aplicação, ou ainda controle de rotas, através de uma interface de comando. Essa interface, chamada Artisan CLI, permite também a definição de arquivos como *migrations* e *models*. Esses arquivos são fundamentais para o Laravel, uma vez que o *migrations* é o modelo usado pelo *framework* para definir as regras de construção do banco de dados através do PHP.

Em adição, para a comunicação com banco de dados, o Laravel possui por padrão o Eloquent, o qual aplica o Design Pattern ActiveRecord, em que cada tabela de banco de dados é representada no código através de uma classe *Model*, utilizada para interagir com essa tabela. Os *Models*, permitem a consulta de dados em suas tabelas, bem como, operações de inserção, atualização e exclusão (DOUGLAS; MARABESI, 2017).

O PHP não oferece suporte nativo a programação assíncrona, mas há algumas bibliotecas que implementam recursos assíncronos, como o ReactPHP (ALEY, 2017). As bibliotecas, que oferecem os recursos assíncronos, são limitadas, quando comparadas ao Node.js. Esta limitação se dá principalmente devido ao código arbitrário que exige recursos multitarefa ou *multithreading* para gerenciamento dos serviços.

4.1.2.1 ReactPHP

O ReactPHP modelado após o Node.js, tornou-se a referência para as bibliotecas assíncronas em PHP (ALEY, 2017). Sua construção fornece uma implementação de funções de aguardo, chamadas promessas, que são baseadas no modelo JavaScript (usado pelo Node).

ReactPHP é uma biblioteca de baixo nível construída para oferecer aos desenvolvedores PHP uma solução orientada a eventos. O EL do ReactPHP, oferece soluções como: Abstração de fluxos, resolvedor de *DNS* assíncrono, cliente/servidor de rede, cliente/servidor *http*, interação com processos (REACTPHP, 2017).

Outra característica importante do ReactPHP é a capacidade de integração com inúmeras outras bibliotecas, oferecendo seus recursos e possibilitando a construção de sistemas cliente/servidor de forma completamente assíncrona e baseado em eventos. Além de contar com uma documentação detalhada de funcionamento, e com uma comunidade ativa, auxiliando aos desenvolvedores.

4.1.3 Comparação

Node.js possui de forma nativa a orientação a eventos, com a computação assíncrona, e sem o uso de *threads*. Isso oferece ao Node.js uma capacidade maior de se adequar a arquitetura, sem a necessidade de bibliotecas adicionais ou tempo adicional para implementar a conexão entre elas.

O *framework* Laravel possui como vantagens a simplicidade e facilidade de desenvolvimento, e é muito bem estruturado através de boas práticas de orientação a objetos e padrões de projetos como, por exemplo, o Eloquent, além de oferecer uma conexão com outras bibliotecas.

A opção escolhida para o desenvolvimento deste projeto foi o *framework* Node.js, versão 8.11.3 por conta do conhecimento prévio da plataforma somado a sua operação que permite uma maior escalabilidade e requer menor integração entre bibliotecas diferentes. Sua capacidade de ser assíncrono, permitindo com que a aplicação não fique parada aguardando a finalização de uma requisição, e por fim, a possibilidade de trabalhar com *servidor* e *cliente* utilizando JavaScript, o que permite uma maior facilidade e produtividade no desenvolvimento deste projeto, uma vez que é utilizada apenas uma linguagem de programação.

4.2 CLIENTE

A escolha do Node.js como servidor, contribui para a escolha do lado cliente, não oferecendo uma escolha fixa, mas com características que possam contribuir tanto em funcionalidade quanto no desenvolvimento do projeto. Desta forma, os *frameworks* escolhidos para o lado cliente, além de amplamente empregados, usam JavaScript como linguagem base de desenvolvimento e podem ser facilmente conectadas ao Node.js

Os *frameworks* escolhidos para comparação foram, como mencionados anteriormente, o Angular e React.

4.2.1 Angular

O Angular é uma biblioteca desenvolvida por engenheiros do Google, que decidiram por cria-lo com intuito de facilitar manutenção de alguns de seus serviços Web. Originalmente, o trabalho de 18 mil linhas de códigos foi reduzido para 1.5 mil linhas, representando aproximadamente 91% de redução de linhas de código (SESHADRI; GREEN, 2014).

Esse ganho foi possível devido, principalmente, a grande modularidade e reusabilidade aplicáveis. A partir da ideia, os engenheiros se dedicaram a criar um método simplificado para o desenvolvimento de aplicações Web. O Google Feedback foi o projeto inicial a usar o Angular, e também a referência para estudos de funcionamento e uso de um *framework* JavaScript da visão de desenvolvedores (SESHADRI; GREEN, 2014).

Em funcionamento, o Angular estrutura a aplicação de forma próxima ao MVC. Os dados são apresentados, em sua maioria de forma direta, através de estruturas JSON, nomeados modelos. A interface com o usuário corresponde à visão. A regra de negócios determina quais partes do modelo serão ou não renderizadas. As regras, por sua vez, são descritas em controladores, que correspondem às lógicas aplicadas de funcionamento do sistema.

Podem ser vistas como vantagens dessa abordagem: Separação entre as camadas da aplicação, deixando o desenvolvimento com maior usabilidade e manutenibilidade. Os controladores não fazem referências diretas às visões, assim, tornando-o independente e permitindo testes rápidos e fáceis sem a necessidade de instanciar um domínio.

É importante ressaltar, que o Angular segue quatro regras, que devem ser aplicadas no desenvolvimento para manter aplicações de grande porte, sem a perda de eficiência e facilidade. Essas regras podem ser observadas no Quadro 3.

Quadro 3 – Quatro princípios do Angular.

Princípio	Descrição
Data-Binding	O Angular trabalha com *data-binding*. Isso permite que os dados apresentados em tela (Linguagem de Marcação de Hipertexto - HTML) possam ser facilmente manipulados pelo controlador. Além da manipulação dos dados para a tela, o Angular permite o uso bidirecional, garantindo que as alterações em tela também sejam acessível em tempo real pelo controlador. Desta forma, um dado terá o mesmo valor mantido e observado por tela e controlador.
Declaração de chamadas	Aplicações Web podem ser construídas a partir de uma única página, os códigos de HTML são alterados por uma linguagem como JavaScript. Por meio de diretivas, o Angular usa um paradigma declarativo. Permitindo que as inserções sejam facilmente associadas a lógica que as chamou.
Controle de dependências	O *framework* conta com um sistema completo de injeção de dependências, que permitem a solicitação de ferramentas adicionais a outros controladores ou serviços, sem a necessidade de instanciá-las. Desta forma, o controlador requisitante não precisa das declarações de como construir suas dependências e as mesmas são iniciadas de imediato, uma vez que são explícitas.
Extensível	Cada diretiva é, idealmente, independente e permite a adição de novas funções para o desenvolvedor, que podem variar de controle de eventos de cliques até criação de estruturas e estilos. O *framework* também conta com uma Interface de Programação de Aplicativos (API) para a expansão das diretivas comuns e inserção de novos modelos.

Fonte – (SESHADRI; GREEN, 2014)

4.2.2 React

React é uma biblioteca JavaScript desenvolvida pelo Facebook, a qual é utilizada para renderizar *views* (por exemplo, páginas HTML) dinamicamente, baseado em algum estado, o qual está muitas vezes em forma de dados. React então atualiza essas *views* geradas, sempre que o estado original mudar(EdgeCoders) (REACT, 2018).

Essas alterações serão refletidas em algum lugar, por exemplo, com a recriação do HTML para o Modelo de Objeto de Documentos (DOM) do navegador. Com React, não é preciso se preocupar em como refletir as mudanças de estado, pois o React simplesmente irá reagir às mudanças e atualizar as *views* necessárias. Além disso, o React também conta com modos de melhorar estruturas como laços e condicionais, desde que seguido algumas de suas regras

(FEDOSEJEV, 2015).

Diferente do Angular que oferece os componentes necessários para quase todas as operações de uma aplicação do lado cliente, o React se preocupa apenas com a *view*, trabalhando com o conceito de componente que recebe propriedades, gerencia estados e apresenta um retorno virtual do DOM. A forma como o React trabalha com esses componentes é estruturando a representação do HTML em objetos.

Um dos problemas que o React buscou resolver é a comunicação bidirecional, que pode tornar-se complexa para depuração. Isso ocorre, porque algumas alterações tem efeito cascata por todo o código, problema que tente a ocorrer com o aumento de complexidade e tamanho das funções. Assim, o React pode ser uma solução adequada para fluxos mais complexos ou que exigem muitos passos. (FEDOSEJEV, 2015)

O React utiliza dois conceitos principais, ele é declarativo e baseado em componentes. A parte declarativa refere-se a capacidade de projetar *views* independentes para cada estado da aplicação, sendo essas renderizadas e atualizadas de forma única. Já para o uso de componentes, cada componente tem seu gerenciamento de estados que por fim resultaram no conjunto estruturado das *views* (CECHINEL *et al.*, 2017).

Com essas duas características e uso do React é possível reutilizar um único componente em múltiplos lugares, com diferentes estados e propriedades, e cada componente pode conter outros componentes. Todos os componentes no React possuem um estado que muda com o passar do tempo, e o React toma conta de atualizar as *views* dos componentes quando seus estados mudarem.

4.2.3 Comparação Angular e React

O React se destaca pela agilidade de atualizar o conteúdo dentro de uma *view*, sendo muito utilizado por sites que têm conteúdos dinâmicos em constante mudança e que utilizam muitos dados, como, por exemplo, o Instagram e Facebook. Outro ponto é que por se tratar de uma biblioteca, é possível utilizá-lo como componente de visualização de outras estruturas, como por exemplo o Angular.

O Angular, por outro lado, é considerado um *framework* completo. Ele contém tudo que é necessário para o desenvolvimento de um aplicativo *client-side*, com uma estrutura geral criada para aplicativos CRUD (*Create, Read, Update, Delete*).

Devido ao projeto não utilizar de páginas com conteúdos em constante mudança e que utilizam muitos dados, e ao fato de Angular ser um framework completo para o desenvolvimento *front-end*, com destaque para a criação de aplicativos com muitas páginas semelhantes envolvendo grande parte do projeto, optou-se pelo uso do Angular como ferramenta de desenvolvimento *client-side*. Para o desenvolvimento foi usada a versão 1.7.0 do Angular, uma vez

que foi encontrado maior número de materiais de apoio e conhecimento para a mesma.

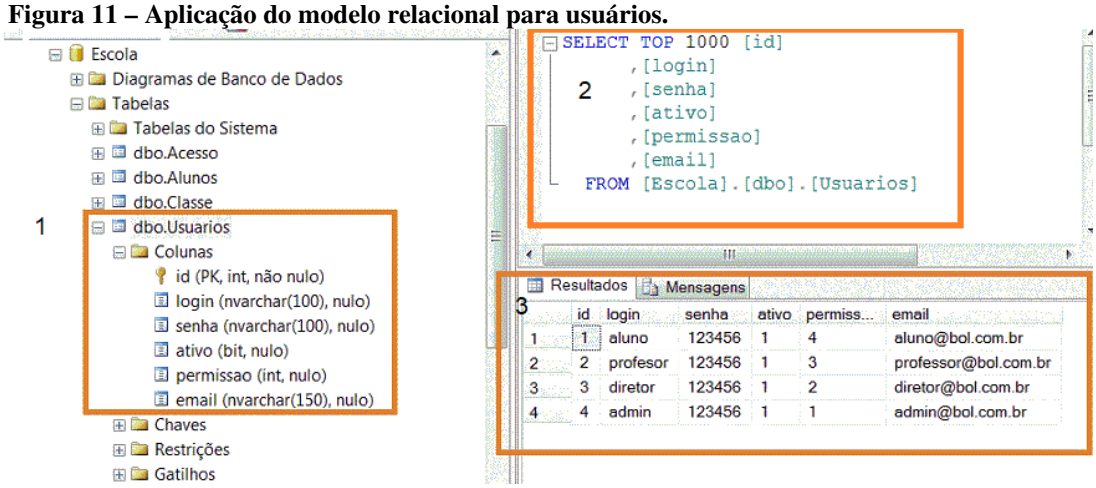
4.3 BANCO DE DADOS

Para a definição entre os inúmeros bancos de dados do mercado, foram comparados dois dos mais usados no mercado de sistemas, sendo cada um único entre si, devido as características. O Microsoft SQL Server é um banco relacional, enquanto o MongoDB é não relacional.

O modelo relacional pode ser visto de três aspectos, a estrutura, integridade e manipulação dos dados (SILBERSCHATZ; KORTH; SUNDARSHAN, 2016). Que são descritos por Elmasri e Navathe (2005) como uma relação de valores, a qual define o modelo. Essa relação é construído por tuplas, que são as linhas, os atributos, que são representados pelas colunas e a tabela em si representa a relação.

A Figura 11 demonstra 3 conceitos aplicados ao modelo relacional. Cada atributo é definido por um tipo (string, inteiro, ...), como apresentado na parte 1 da Figura 11. A parte 2 demonstra o acesso aos dados via comandos SQL, que permitem operações complexas entre relações diferentes, ou simples acesso a algum dado, como esse exemplo. A parte 3 é a apresentação dos dados, no qual cada tupla representa, neste caso, um usuário diferente.

Figura 11 – Aplicação do modelo relacional para usuários.



Fonte: Autoria própria

Um dos requisitos, no modelo relacional, é a necessidade que a estrutura do banco de dados deve ser projetada, os esquemas das tabelas devem ser definidos, para permitir o uso da relação. Também devem ser adicionadas novas tabelas para novos tipos de informações, criando ou não uma relação entre ela e as demais se existir alguma referência. Por exemplo, uma tabela de telefones, pode ter uma referência indicando a qual usuário os mesmos pertencem.

O modelo não relacional, também chamado de NoSQL, foi desenvolvido como alternativa ao modelo relacional, visando características que o modelo relacional não atende como desejado. Os bancos NoSQL foram desenvolvidos da necessidade da web moderna, como a ne-

cessidade de alta capacidade de armazenamento, eficiência no acesso e gravação dos dados e escalabilidade (TIWARI, 2011).

A classificação dos bancos NoSQL é dado conforme o modelo de armazenamento dos dados. Sendo basicamente 4 categorias, orientado a documentos, orientado a colunas, grafos ou chave/valor, sendo o orientado a documentos um dos mais populares (POPESCU, 2010).

No modelo orientado a documentos, são armazenados, recuperados e gerenciados dados semi-estruturados. Cada entrada é chamada de documento, que é referenciado por uma chave única. Um tipo comum usado para armazenamento é o JSON, que é um arquivo onde é possível armazenar chaves, valores e hierarquias (CHODOROW, 2013), como demonstra Figura 12.

Figura 12 – Informações de um usuário apresentadas como JSON

```
{
    "login": "aluno",
    "id": 1,
    "senha": 123456,
    "ativo": 1,
    "permissao": 4,
    "email": "aluno@teste.com"
}
```

Fonte: Autoria própria

4.3.1 SQL Server

O Microsoft SQL Server é um sistema gerenciador de banco de dados (SGBD) relacional que suporta uma ampla variedade de processamento de transação, *business intelligence*, e aplicações analíticas em ambientes corporativos de TI. O sistema é fornecido pela Microsoft, e foi apontado como um dos melhores SGBD no mercado (OZGUR *et al.*, 2017).

O sistema é distribuído em várias versões e edições, podendo o SQL Server ser utilizado em projetos de diversos tamanhos. Porém algumas versões do SGBD são pagas, enquanto outras livres, como a *express*, possuem limitações. Por exemplo, limite de tamanho da base de dados, número de processadores e outras.

Em compensação aos preços pagos, o SQL Server apresenta várias características muito procuradas em SGBDs. Entre elas, pode-se citar, alta disponibilidade, recuperação de desastres, segurança e ferramentas de gerenciamento robustas (OZGUR *et al.*, 2017). Em 2016, a empresa (Microsoft) anunciou o desenvolvimento de uma versão Linux para o SGBD, que tinha como maior barreira a limitação de S.O.

4.3.2 MongoDB

MongoDB é um banco de dados de código aberto, gratuito, de alta performance, sem esquemas e orientado à documentos. Lançado em fevereiro de 2009 pela empresa 10gen, foi escrito na linguagem de programação C++, o que o torna portável para diferentes S.O (BATISTA et al., 2013).

Por ser orientado à documentos JSON, muitas aplicações podem modelar informações de modo muito mais natural, pois os dados podem ser aninhados em hierarquias complexas e ainda serem indexáveis e fáceis de buscar, igual ao que já é feito em JavaScript.

Neste tipo de banco (*document-based* ou *document-oriented*), tem-se coleções de documentos, nas quais cada documento é auto-suficiente, ele contém todos os dados que possam precisar, ao invés do conceito de não repetição e as chaves estrangeiras, presente no modelo relacional (CUNHA, 2011).

A ideia é não utilizar *joins* (relacionamento entre tabelas), pois estes são prejudiciais à eficiência das *queries*. A modelagem da base deve ser feita de forma que a cada *query*, será feita uma única busca no banco e com apenas uma chave primária é selecionado tudo que for necessário.

O MongoDB é visto como um dos mais bem sucedidos bancos NoSQL. Tal fato, devido à sua linguagem de consulta de alta performance, a qual baseia-se em documentos e na facilidade de importar os dados de um banco relacional.

4.3.3 Definição SGBD

A análise de requisitos mostrou uma quantidade elevada de páginas de cadastros e relacionamento entre os dados coletados. Além dos cadastros, serão gerados diversos relatórios e estatísticas, exigindo comandos *sql* específicos. Desta forma, o SGBD SQL Server, apresenta um valor inicial maior. Embora alguns dos produtos sejam pagos, as versões gratuitas oferecem recursos suficientes para atendimento de pequenas organizações.

O motivo que impulsionou o uso do SQL Server e não outro SGBD de licença gratuita foi a disponibilidade de uma chave de uso pela empresa Microsoft para uso deste projeto.

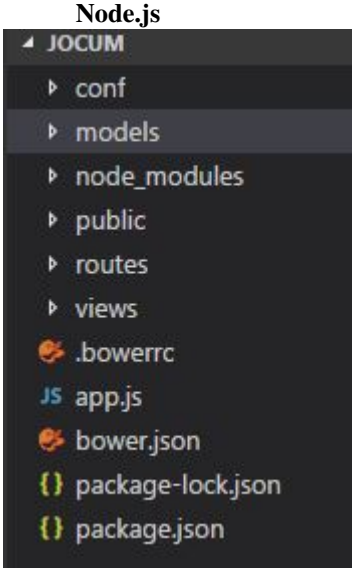
5 DESENVOLVIMENTO DO SISTEMA E MODELO BÁSICO

Neste capítulo são apresentados os passos realizados para o desenvolvimento prático do projeto, incluindo a construção do servidor em Node.js, a definição do sistema para o cliente e a modelagem do banco de dados.

5.1 SERVIDOR NODE.JS

O desenvolvimento do servidor partiu da estrutura básica do Node.js, com a definição dos arquivos de origem e extensões. A esses arquivos foram adicionadas as pastas para armazenamento de modelo, configurações, rotas e outros, como apresentado na Figura 13.

Figura 13 – Construção do servidor em Node.js



Fonte: Autoria própria

A seguir serão abordados os itens definidos na construção do servidor e suas funções. Com exceção das rotas, modelos e configurações específicas a cada parte da implementação, todo o servidor foi implementado no ciclo 1 de desenvolvimento.

5.1.1 package.json

O package.json é considerado o ponto inicial de um projeto em Node. Através deste arquivo é definido o projeto, adicionando nome, licença, autores, versão, e outras informações. Neste arquivo também são informadas todas as dependências que serão usadas pelo servidor.

O package.json também pode listar alguns *scripts* que podem ser usados para criar

comandos de acesso, ou mesmo definir o ponto de início do servidor. Uma das vantagens de usar um sistema Node, é o grande número de bibliotecas que podem ser utilizadas para tratar funções comuns.

Para este projeto foram adicionadas várias dependências, como Express. A estrutura dessa biblioteca é simples, e tem o objetivo o desenvolvimento flexível da estrutura da aplicação. O Express oferece também recursos como roteamento de Localizador Padrão de Recursos (URL), controle de sessão, gerenciamento de requisições, que são amplamente usados no lado servidor (VLADUTU, 2014).

As bibliotecas adicionais tem funções menos abrangentes, sendo aplicadas para comunicação com banco de dados (MS SQL), para realizar tratamento do corpo das requisições (*cookie* e *body-parser*). Para este projeto também foram adicionadas as bibliotecas *grunt* que são destinadas a proteger o código fonte.

5.1.2 Arquivo origem

O arquivo origem (app.js) representa o ponto inicial do servidor. No app.js são definidas as regras as quais o servidor irá operar, sendo elas as configurações de execução ou conexões com bancos de dados e definição das rotas e outros acessos do sistema.

Para a organização do sistema, foi definido que cada chamada tratada pelo servidor, será tratada por uma rota específica. A rota é definida por qual o estado principal solicitante. A Figura 14 apresenta um exemplo de como a chamada "localhost/usuario", seria tratada pela rota login, definida no arquivo de origem.

Figura 14 – Chamada da rota de usuário no servidor.

```
const usuario = require('./routes/usuario');
app.use(home + '/usuario', usuario);
```

Fonte: Autoria própria

Desta forma, pode-se concentrar rotas de inserção, alteração ou exclusão em uma rota única, mas limitada a um propósito específico. Além da facilidade em localizar a rota acionada, pode-se facilmente adicionar ou retirar novas rotas, sem impactar o funcionamento das demais.

5.1.3 Rotas

As rotas, dentro do sistema são responsáveis pelo acionamento da função específica a cada evento. Desta forma, cada rota é única e para cada evento disparado pelo cliente, deve existir uma rota no servidor, a qual é responsável por acionar a função e apresentar um retorno.

A Figura 15 apresenta um exemplo de construção dessas rotas. Na figura é possível observar que a rota é apenas uma declaração de qual função será executada, os parâmetros, e como será realizado o tratamento do retorno. Como as chamadas são essencialmente iguais, a variação entre uma rota e outra é dada apenas pelo tipo Transferência de Estado Representacional (REST) sendo tratado e qual função está sendo requisitada.

Figura 15 – Tratamento das rotas referentes ao usuário.

usuario.js

```
1   const express = require('express');
2   const Usuario = require('../models/usuario');
3   let router = new express.Router();
4
5   //SEARCH relação de usuários
6   router.get('/relacao/usuario', function(req, res) {
7     let usuario = new Usuario();
8     usuario.buscarTodosUsuarios(req.body.data, req.headers)
9       .then(enviarResultado.bind(null, res))
10      .reject(enviarErro.bind(null, res));
11  });
12
13  //GET relação de usuários
14  router.get('/usuario', function(req, res) {···
19  });
20
21  //POST para inclusão de usuário
22  router.post('/usuario', function(req, res) {···
27  });
28
29  //PUT para edição de usuário
30  router.put('/usuario', function(req, res) {···
35  });
36
37  //DELETE para edição de usuário
38  router.delete('/usuario', function(req, res) {···
43  });
```

Fonte: Autoria própria

5.1.4 Modelos

Um modelo para o Node.js é uma representação de uma relação do banco de dados. Nessa representação são definidas as regras de negócio, centralizando validações e ajustes necessários aos dados. Essas regras ficam definidas por funções, que por sua vez, são acionadas pelas rotas.

Após a validação, a função então requisitará a outra o envio ao banco ou rejeitará os

dados. Embora a maioria dos modelos opere deste modo, como na Figura 16, existem ainda dois modelos especiais, o conectaDB e o modelo.

Figura 16 – Modelo de usuário.

Usuario.js

```
const modelo = require('./modelo');
class Usuario {
  constructor(){}

  buscarTodosUsuarios(data, metodo, headers) {
    montarResposta('usuario', data, metodo, headers);
  }
  buscarUsuario(recurso, data, metodo, query) {···
  }
  salvarNovoUsuario(recurso, data, metodo, headers) {···
  }
  salvarUsuario(recurso, data, metodo, headers) {···
  }
  deletarUsuario(recurso, data, metodo, headers) {···
  }
  montarResposta(servico, parametros, metodo, headers) {
    modelo.montarResposta(
      servico,
      parametros,
      metodo,
      headers,
      (resposta) => {
        resolve({status: 200, mensagem: resposta});
      },
      (erro) => {
        resolve({status: 500, mensagem: erro});
      });
  }
};

module.exports = Usuario;
```

Fonte: Autoria própria

A chamada de envio ao banco de dados, acontece através da chamada da função montarResposta. Serão enviados:

- recurso (serviço) - define qual arquivo da pasta conf será acessado
- parâmetros - dados enviados ao banco
- método - qual operação será realizada
- cabeçalhos (*headers*) - possuem as informações de sessão para validação de usuário

5.1.4.1 conectaDB

O modelo conectaDB tem o objetivo de centralizar todas as ações de conexão com banco de dados. O modelo, gerencia novas conexões e armazena as realizadas em um *pool*, para que não sejam necessários reconectar a cada operação. O banco de dados pode ser selecionado de acordo com o nome da conexão, passada através dos headers, ou como nome padrão, estabelecido através do arquivo conf.json na pasta conf.

Essa opção permite que o servidor seja capaz de se comunicar com bases diferentes. Com essa característica é possível que um servidor atenda dois clientes distintos, com bases diferentes. Desta forma é possível expandir o uso do projeto, por exemplo em outras localidades onde a Jocum esteja presente, sem que os dados entre os usuários entrem em conflito, ou perca-se a privacidade necessária.

A Figura 17 apresenta o principal ciclo da classe, onde é iniciado a conexão com banco de dados. Para construção da conexão, são necessários os parâmetros (servidor, base, usuário, senha) podem ser definidos de duas formas. A primeira, como já mencionado, através do arquivo de configuração, e a segunda via variáveis de ambiente. As variáveis de ambiente consistem em configurações de execução, que permitem acesso a uma informação comum por todo o servidor. Sistemas desenvolvidos em Node.js são executados normalmente em estruturas chamadas *containers*, que consistem em instâncias da execução.

Programas, como Docker, possibilitam o gerenciamento do ambiente e suas variáveis de controle. Tais configurações podem ser úteis, uma vez que pode-se realizar alteração de endereço do banco de dados sem a necessidade de alterar o código fonte. Além das variáveis, é usado para a conexão a dependência MS SQL, que é responsável por realizar o intermédio entre o banco de dados e o Node.js.

Uma vez estabelecida a conexão, essa é armazenada em um objeto, que será usada nas chamadas posteriores para realizar a troca de mensagens. A primeira conexão é iniciada junto ao sistema, sendo definida no arquivo app.js.

A parte de conexão foi idealizada para atender a dois princípios. O primeiro sendo manutenção, uma vez que todas as conexões ficam centralizadas, qualquer alteração pode ser realizada de forma ágil em um único ponto do sistema. O segundo princípio e a alteração do banco de dados. Embora o sistema tenha sido desenvolvido para se comunicar com SQL Server, a centralização das conexões permite também que ao mudar o sistema para comunicar-se com outro SGBD, o servidor seja impactado em poucos locais. Esses princípios também se aplicam ao modelo.js.

Figura 17 – Estabelecimento de conexão com banco de dados

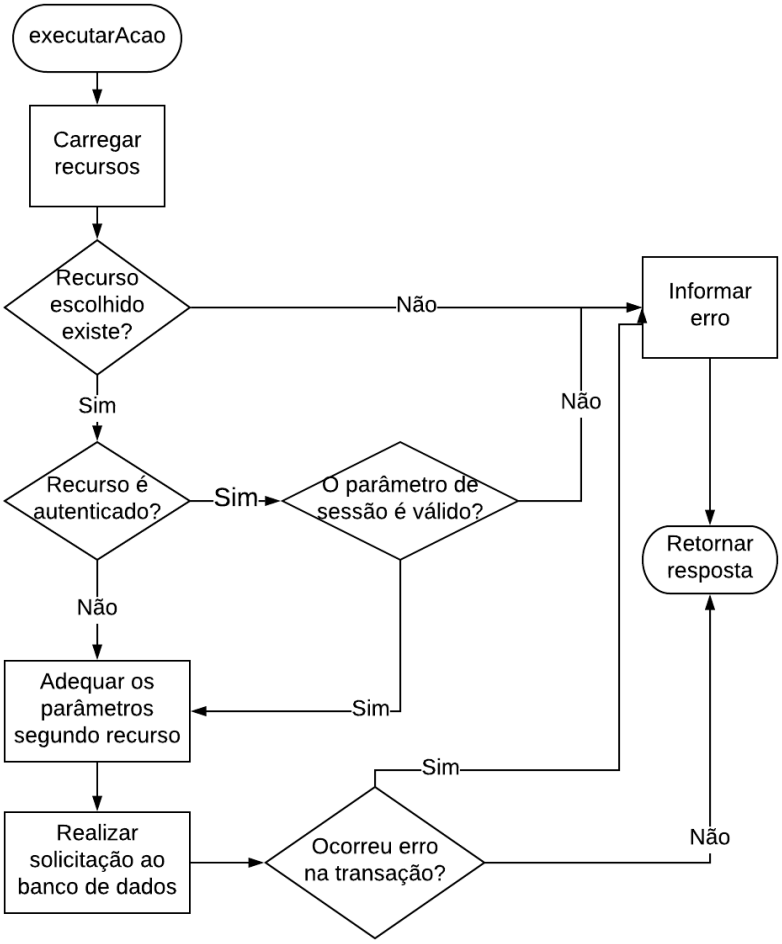
```
let vConfiguracaoConexao = conectaBD.montarConfiguracaoConexao(
    confJson.servidor || process.env.SERVIDOR,
    confJson.base || process.env.BASE,
    confJson.usuario || process.env.USUARIO,
    confJson.senha || process.env.SENHA
);

repoBD.confConexoes['mainConnection'] = vConfiguracaoConexao;
repoBD.confConexoes['mainConnection'].status = 0; // criando conexao...

console.log('conectaBD: criando mainConnection server: ' + vConfiguracaoConexao.server + ' database: ' +
repoBD.conexoes['mainConnection'] = new mssql.Connection(vConfiguracaoConexao, function(err) {
    if (err) {
        console.error('conectaBD: erro criando mainConnection ' + err);
        repoBD.confConexoes['mainConnection'].status = -1; // conf ok sem conexao
        reject(err.message);
    } else {
        repoBD.confConexoes['mainConnection'].status = 1; // conexao pronta
        console.log('conectaBD: mainConnection ok status ' + repoBD.confConexoes['mainConnection'].status
        confJson.conexoes = confJson.conexoes || {};
        for (let chave of Object.keys(confJson.conexoes)) {
            repoBD.confConexoes[chave] = confJson.conexoes[chave];
            repoBD.confConexoes[chave].status = -1; // conf ok sem conexão
        };
        resolve();
    }
});
```

Fonte: Autoria própria

5.1.4.2 modelo

Enquanto o conectaDB é responsável pelas conexões com o banco, o modelo é responsável por todas as operações de manipulação de dados. A principal função do modelo.js é a *executarAcao*, que é responsável por controlar os recursos, que são definidos por aquivos na pasta conf. Embora a função possa ser chamada diretamente, devido a sua complexidade, dentro das classes comuns do modelo, a mesma é executada através de uma função anterior chamada *montarResposta*.

Figura 18 – Fluxo de execução para operações com banco de dados



Fonte: Autoria própria

O fluxograma apresentado na Figura 18, demonstra a sequência de passos usados na função *executarAcao* uma ação com banco de dados. O primeiro passo é a requisição, que deve apresentar diversos parâmetros, que serão usados para validar usuário, seleção dos parâmetros, recurso e funções de retorno. Em seguida, o sistema irá verificar se o recurso solicitado está carregado junto a execução. Todos os recursos são armazenados no momento de início do sistema, sendo que alterações nos recursos demandam de reinicialização do servidor.

Caso o recurso exista, é então verificado se o mesmo exige autenticação. Essa autenticação é dada através do código da sessão, estabelecida em cada acesso ao sistema. Essa validação é opcional, sendo definida individualmente para cada recurso. Essa visa permitir que outros sistemas possam acessar determinados recursos sem a necessidade de cadastro de usuário, assim como deixar recursos protegidos, a escolha do cliente.

Apenas após a verificação do recurso e validação é realizada o envio ao banco. Neste ponto, foram adicionados possíveis ajustes aos parâmetros da requisição. Para casos nos quais seja necessário o envio do nome ou código do usuário, ou código da sessão, esses podem ser declarados através do recurso.

Também é permitido retirar parâmetros, através da definição de campos calculados. Desta forma, não é necessário manipular cada parâmetro da requisição em tela, para cada chamada, pois podem ser facilmente controlados através do recurso, no servidor.

5.1.5 Arquivos Conf

A pasta de recursos (conf), é destinada a dois tipos de arquivos. O arquivo de configuração do servidor, onde são armazenados os dados de conexão com banco de dados. Além deste, são armazenados os arquivos com a definição dos recursos disponíveis com ao banco de dados.

Não são adicionados comandos SQL diretamente no servidor, de modo que cada chamada ao banco deve corresponder a uma *procedure*. Estas são indicadas nestes arquivos de configuração, em JSON, como na Figura 19.

Figura 19 – Recursos disponíveis para usuário no servidor

Fonte: Autoria própria

Observa-se que devem ser especificados os campos "recurso"e "acoes". O recurso corresponde ao nome pela qual será referenciado o recurso dentro do servidor. As ações, serão definidas pelas chamadas REST. Não é obrigatório a definição de todos os recursos para que sejam executados.

5.1.6 Opções Adicionais

A pasta node_modules é construída automaticamente através do *download* das dependências. Caso as dependências não tenha sido baixadas e estejam sendo usadas em algum momento, ocorrerá um erro ao iniciar o servidor. Caso semelhante ao arquivo package-lock.json, que referencia as dependências instaladas com o servidor. pasta *public* é destinada aos arquivos do cliente, sendo abordado em um seção específica. Os arquivos bower.json e .bowerrc também

são destinados à parte cliente.

5.2 *FRONT-END* COM ANGULAR

O desenvolvimento com Angular teve o objetivo primário de construir um sistema no qual a apresentação pudesse ser feita de maneira ágil e de fácil manutenção. Para alçancar o objetivo, foram criadas diretivas e serviços capazes de gerar os templates usados na aplicação.

Também foram adicionadas bibliotecas para funções comuns. Essas funções são a criação de componentes de datas e calendários, manipulação de requisições HTTP com REST, fontes gráficas e recursos de estilos.

5.2.1 *Templates*

Os *templates* são os documentos de conteúdo visual. É destinado a organização dos arquivos HTML. No projeto, são armazenados os recursos usados por todas as diretivas, como também os arquivos relacionados a cada *view*. Cada *view* ou *template* é acionado por um ou mais controladores. Desta forma, as páginas HTML são acionadas a medida que os controladores mudam de estado.

5.2.2 Diretivas

As diretivas para o Angular, são extensões da linguagem HTML. Essa extensões tem o objetivo de oferecer novos elementos. Alterando recursos já existentes ou criando novos. Desta forma, é possível usar diretivas para manipular o DOM, associar funções JavaScript a eventos ou adicionar novos HTMLs. Neste projeto, foram adicionados alguns recursos para criação de telas a partir de arquivos de configuração.

5.2.2.1 Editável

A diretiva "editável"foi construída para definir os componentes da tela. Através da diretiva, é possível definir botões, *labels*, caixas de texto, *radiobuttons*, caixas de seleções, *links*, *checkbox* e outras opções, definindo quase todas as entradas mais usadas em páginas web. Essa medida permite a definição de algumas características para o lado cliente da aplicação.

A primeira característica é a padronização de tela, uma vez que os campos construídos a

partir da diretiva respeitarão suas regras, tornando o sistema visualmente consistente. O segundo ponto é a capacidade de criar qualquer campo definido por ela, em qualquer tela do sistema, chamando uma única *tag* e definindo seus parâmetros, inclusive o uso da mesma em outras diretivas, como a *autoform*.

Todos os campos são construídos a partir de diversas opções, que podem ser definidas segundo uso. A Figura 20 apresenta exemplos dos usos desta diretiva, sendo um campo texto e outro tipo senha. Segundo as configurações, esses campos estarão sempre habilitados para edição, tem um título especificado no campo, tem eventos tratados por funções específicas, e é identificado qual variável e qual é o controlador responsável por esta informação.

Figura 20 – Exemplo de aplicação da diretiva editável

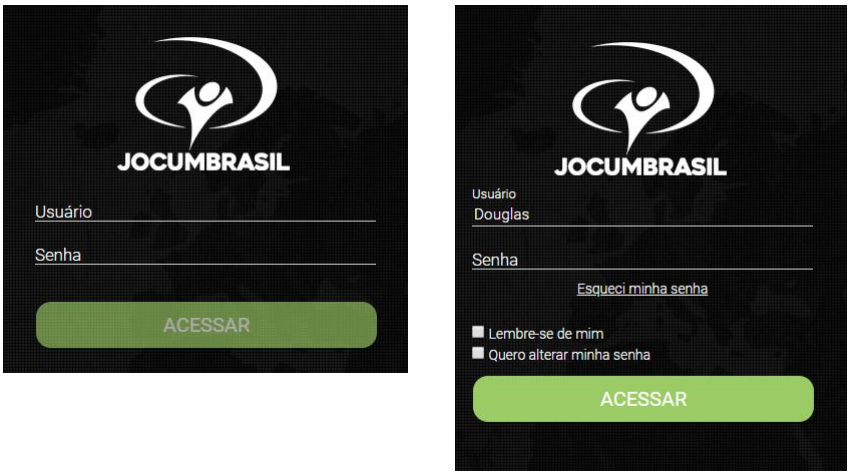
```
<vs-editavel3 tipo="text"
  editavel="sempre"
  modelcadastro="cadastro"
  ng-model="cadastro.registro.usuario"
  ngChange="modelcadastro.verificarUsuarioCliente()"
  placeholder="Usuário"></vs-editavel3>
<vs-editavel3 tipo="password"
  editavel="sempre"
  modelcadastro="cadastro"
  ng-model="cadastro.registro.senha"
  ng-keypress="cadastro.verificarEvento($event)"
  placeholder="Senha"></vs-editavel3>
```

Fonte: Autoria própria

Observa-se que para os exemplos da Figura 20, o ganho na quantidade de informações necessárias não é tão significativo quanto um *input* tradicional, resultando na tela da Figura 21. Alguns dos pontos a favor do uso da diretiva são mais visíveis com a declaração de campos tipo combo, onde são necessários apenas três configurações para apresentar todas as opções, ou mesmo no fato de todas as configurações de estilo estarem adicionadas uma única vez dentro da mesma. Embora a diretiva apresente vantagens em seu uso, seu principal intuito é quando usada juntamente a diretiva *autoform*.

Figura 21 – Exemplo de tela de acesso, com a Figura 21(a) representando o estado padrão e a Figura 21(b) representando as alterações dinâmicas.

(a) **(b)**



Fonte: Autoria própria

5.2.2.2 Autoform

Um dos objetivos do sistema desenvolvido, foi a criação de telas de forma dinâmica e rápida. Para isso foi desenvolvida a diretiva de *autoform*. Todas as telas construídas a partir dessa diretiva podem ser especificadas através de um arquivo JSON, como exemplo da Figura 22.

Figura 22 – Exemplo de configuração para tela dinâmica com diretivas.

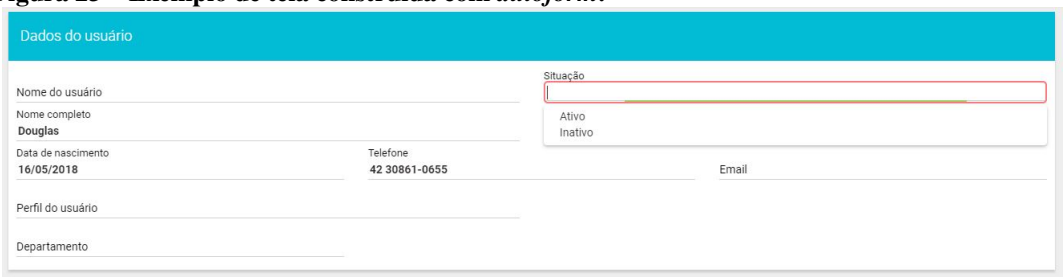
```
{
  "servico": "usuario",
  "campos": [
    {…
    },
    {
      "nome": "usr_situacao",
      "descricao": "Situação",
      "tipo": "select",
      "selectOpcoes": "situacaoUsr",
      "selectChave": "valor",
      "selectTexto": "descricao",
      "obrigatorio": true
    }
  ],
  "consulta": {
    "campos": ["usr_nome"]
  },
  "relacao": {
    "registrosPagina": 8,
    "campos": ["usr_nome",
               "usr_situacao"]
  },
  "autoForm": {
    "grupos": [
      { "descricao": "Dados do usuário",
        "linhas": [{ "campos": [{"nome":"usr_nome"},{"nome":"usr_situacao"}]}]
      }
    ]
  }
}
```

Fonte: Autoria própria

São especificados no JSON, o nome do serviço, os campos e apresentação dos dados. É possível também adicionar um campo de busca e escolher qual o parâmetro que será buscado. As regras de apresentação podem definir a exibição de uma relação de informações referente ao serviço (diretiva Relação), com a escolha dos campos e ordem dos mesmos, além da definição de quantos registros serão expostos, criando paginações.

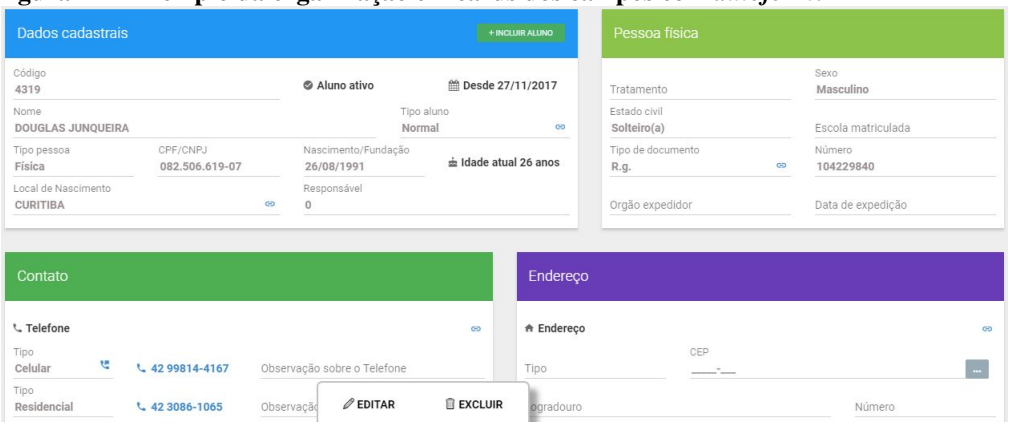
A apresentação individual das informações, com os campos é dada através especificamente da definição *autoform*, onde a tela é organizada em *cards*, como demonstrado através das Figuras 23 e 24, que possuem suas propriedades, como tamanho, cor, título, e os campos e sua ordem de exibição. No exemplo da Figura 22, foi criado um *card* de tamanho padrão, de título "Dados do usuário", com os campos de texto nome, e combo Situação, na mesma linha.

Figura 23 – Exemplo de tela construída com *autoform*.



Fonte: Autoria própria

Figura 24 – Exemplo da organização em cards dos campos com *autoform*.



Fonte: Autoria própria

Desta forma, com um único arquivo de configuração, e uma chamada como da Figura 25, é possível construir duas telas. A primeira com a relação do serviço chamado e outra para os campos e detalhes individuais, usada para inserção, edição e exclusão de dados.

Figura 25 – Chamada das diretivas de relação e *autoform* em tela.

```
<vs-relacao modelcadastro="cadastro" ng-if="state.is(cadastro.statePrincipal)">
</vs-relacao>

<div ng-if="state.is(cadastro.statePrincipal+'.detalhe')||state.is(cadastro.statePrincipal+'.detalhe.edicao'
  <vs-auto-form modelcadastro="cadastro">
  </vs-auto-form>
</div>

<div ui-view="conteudo" ng-if="state.includes(cadastro.statePrincipal + '.detalhe.**')">
</div>
```

Fonte: Autoria própria

5.2.2.3 Relação

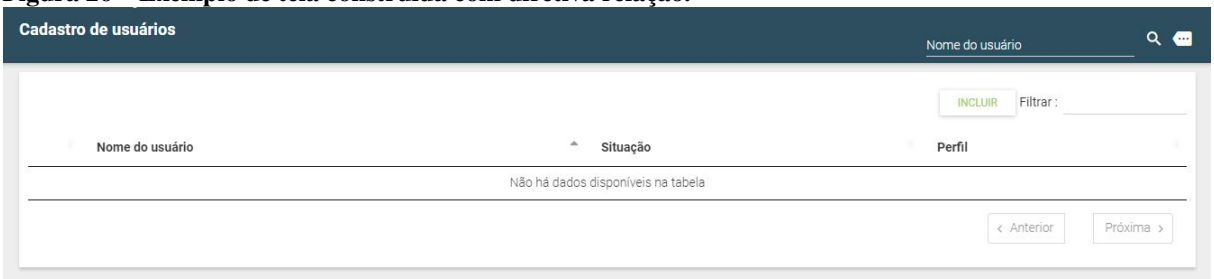
Idealmente todas as páginas de cadastros devem listar todos os registros cadastrados, assim como dados relevantes sobre eles, de forma organizada. A diretiva de relação foi desenvolvida para apresentar essas relações.

Construir relações individuais para cada página demandam de duplicações enormes de código, com gasto de tempo para o desenvolvimento e manutenção. Essencialmente, essas telas representam o mesmo propósito, diferenciando entre si apenas a quantidade de campos apresentados e o conteúdo. Adicionar uma relação torna-se muito mais prática, como demonstra as Figuras 22 e 25, com a especificação da listagem de 8 usuários por página, com as informações de nome e situação.

Assim, ao construir uma diretiva capaz de identificar os campos e tipos que devem ser listados, é possível reduzir o tempo gasto para sua apresentação em poucos passos, obtendo um resultado como o apresentado na Figura 26. A diretiva Relação constrói uma tabela dinâmica, através da leitura do arquivo JSON de serviço, no qual são especificados os campos apresentados, quantidade de registros por página e o tipo de cada campo.

Além dos campos relacionados ao banco, é possível também adicionar botões à diretiva, para apresentar detalhes ou ícones que representem informações. Esses valores devem também ser especificados no arquivo de configuração. Assim como as demais diretivas, a relação é um recurso amplamente usado, garantindo que as relações apresentadas não apresentem grandes diferenças entre si.

Figura 26 – Exemplo de tela construída com diretiva relação.



Fonte: Autoria própria

5.2.2.4 Outros

Foram adicionadas algumas diretivas menores, como (BOOTSTRAP, 2016) e data (ANGULARJS, 2010) que são baseadas em bibliotecas externas. Essas diretivas são usadas no sistema para reproduzir algum comportamento específico para campo, como data ou busca de valores em pontos chave do sistema.

5.2.3 Serviços JS

Serviços para o Angular são funções temporárias, que são injetadas dentro de controladores. Eles têm o objetivo de organizar e partilhar funções dentro do projeto. Os serviços tem duas características: instanciamento apenas quando algum componente requisita e são execuções únicas (*Singleton*), passando apenas a referência para o componente requisitante.

5.2.3.1 *Security*, *Messages* e Configurações

Existem alguns serviços gerais, que são utilidades para controles de variáveis do sistema, como segurança e mensagens de alerta na Figura 27. O serviço *Messages* tem o intuito de apresentar alertas gerais e informações de confirmação ou erro na tela. Assim, o tratamento para apresentação de mensagens é comum para todo o sistema.

Figura 27 – Exemplo de mensagem de erro do sistema.

Fonte: Autoria própria

O serviço de *security* foi implementado para realizar as validações de segurança do sistema. Desde o acesso ao sistema, as validações e armazenamento das variáveis de sessão são armazenadas pelo serviço. No *security* também é definido o estado principal do sistema, e o estado de acesso.

Esses serviços, embora simples, são usados em diversos pontos do sistema. Especialmente quando há acesso à informações que exigem autenticação ou para telas de cadastros, nas quais há muitas mensagens de validação ou confirmação de operação.

O serviço de configuração é fundamental para o funcionamento do *framework*. O serviço tem dois objetivos principais, carregar as opções de segurança para a *view* e carregar as informações da própria *view*.

As opções de segurança são carregadas inicialmente através de uma requisição HTTP, o qual retornará se um usuário possui permissão de visualização da *view* ou informará o erro ao usuário. Caso, o mesmo possua acesso, é carregado as informações do arquivo de configuração da *view*, e a página é exibida dinamicamente.

5.2.4 Serviços de configuração

Os serviços de configuração são destinados a especificação de tela. Através destes é possível definir diversas características da apresentação, como os campos, divisão da tela, tamanho de cada divisão, campos por linha, tipos dos campos e outras.

Esses serviços, definidos por um arquivo JSON, como na Figura 22, armazenam todas as informações de tela. Definições escritas diretamente no arquivo HTML não são lidas pelas diretivas, nem configuradas automaticamente, sendo necessário a chamada de alguma diretiva, ou mesmo especificar campo a campo suas propriedades.

5.2.5 Controladores

Os controladores ou *controllers*, são responsáveis pelo gerenciamento da aplicação. Todo o fluxo de dados apresentado é manipulado em algum ponto através do controlador. Os controladores são instanciados, e anexados ao DOM através de sua referência no arquivo inicial do cliente (index.html).

Cada controlador trabalha com um espaço chamado *$scope* (escopo). O escopo delimita quais variáveis, funções, serviços podem ser acessados e concentra a lógica de negócio destinado ao cliente, como a validação de campos.

O controlador também é responsável por gerenciar os estados possíveis de uma *view*. Por exemplo, a Figura 28, apresenta os estados possíveis para a *view* de usuário, onde são descri-

tos o estado raíz, e o estados filho "detalhe"com outras duas possibilidades, "inclusao"e "edicao".

Os nomes dos estados com informações específicas foram padronizados no sistema, sendo "detalhe"ou "detalhe.(edicao ou inclusao)"de acordo com a situação da página. Isso permite aplicar propriedades de manipulação específicas, por exemplo bloqueando o campo nome do usuário durante a edição.

Figura 28 – Estados possíveis para *view* de usuários.

Fonte: Autoria própria

Para fins de estruturação, cada *view* do projeto possui o próprio controlador, que está descrito em arquivos separados ou junto ao arquivo com controlador do estado raiz. Essa organização permite uma leitura mais rápida e facilita na manutenção do código, uma vez que controladores de *views* complexas podem exigir lógicas extensas.

Figura 29 – Exemplo de controlador simples.

```
JocumWeb.controller("UsuarioCtr", ["$scope", "serviceCadastro","serviceConfiguracao","$state",
  function($scope, serviceCadastro, serviceConfiguracao, $state) {
    $scope.cadastro = {
      configuracao: {},
      registro: {},
      relacao: {},
      filtro: {},
      combos: {},
      parametrosBusca: {},
      statePrincipal: "usuario"
    };
    serviceCadastro.setModelCadastro($scope.cadastro);

    $scope.state = $state;

    // Busca configurações
    serviceConfiguracao.buscarConfiguracoes("usuario").then(function(pConfiguracoes) {
      $scope.cadastro.configuracao = pConfiguracoes.usuario;
    });

  }]);
```

Fonte: Autoria própria

A Figura 29 retrata a construção básica de um controlador. Definido por seu nome, injeção de dependências e a função que o define. Para o sistema, foi adotado o padrão de armazenar as informações em um objeto chamado "cadastro".

As configurações destinam-se a armazenar questões de funcionamento da *view*, como permissões de acesso ao recurso em particular. O registro é responsável pelo armazenamento do dados referentes a uma entidade, no caso, um usuário. O *array* relação é responsável por armazenar a relação completa de uma busca, armazenando neste exemplo todos os usuários, que, posteriormente, serão listados através da diretiva Relação.

São ainda partes importantes do cadastro os "combos", que armazenam os valores de cada campo tipo *select* da *view* usuário. Alguns combos são construídos em tela, enquanto outros são acessados do banco, como lista de nome de usuários para busca. Essa medida foi adotada para que telas com muitas alterações e itens possam ser relacionados e modificados pelo próprio usuário, mas não totalmente para que telas com poucas opções não necessitem a chamada adicional ao banco para opções fixas.

6 RESULTADOS

O propósito deste trabalho foi a criação de uma ferramente de desenvolvimento, que permitisse o desenvolvimento de um sistema completo, para gerenciamento de organizações de assistência social. Essa ferramente buscou ser uma solução capaz de abranger as situações gerais e específicas no desenvolvimento destinado a *web*.

Para tal tarefa, foi usado o Microsoft SQL Server como SGBD, o Node para construção do servidor e o Angular como base do cliente. A avaliação final da plataforma foi dada a partir de três quesitos:

- Tempo de resposta, medindo tempo total para realizar operações do sistema, essa medida pode ser entendida como a influência no nível de frustração do usuário com a demora para a resposta de sua ação.
- Escalabilidade, que é a medida de como o acréscimo de recursos, neste caso *hardware*, afeta o desempenho da aplicação.
- Latência, que representa o tempo mínimo requerido para obter qualquer forma de resposta.

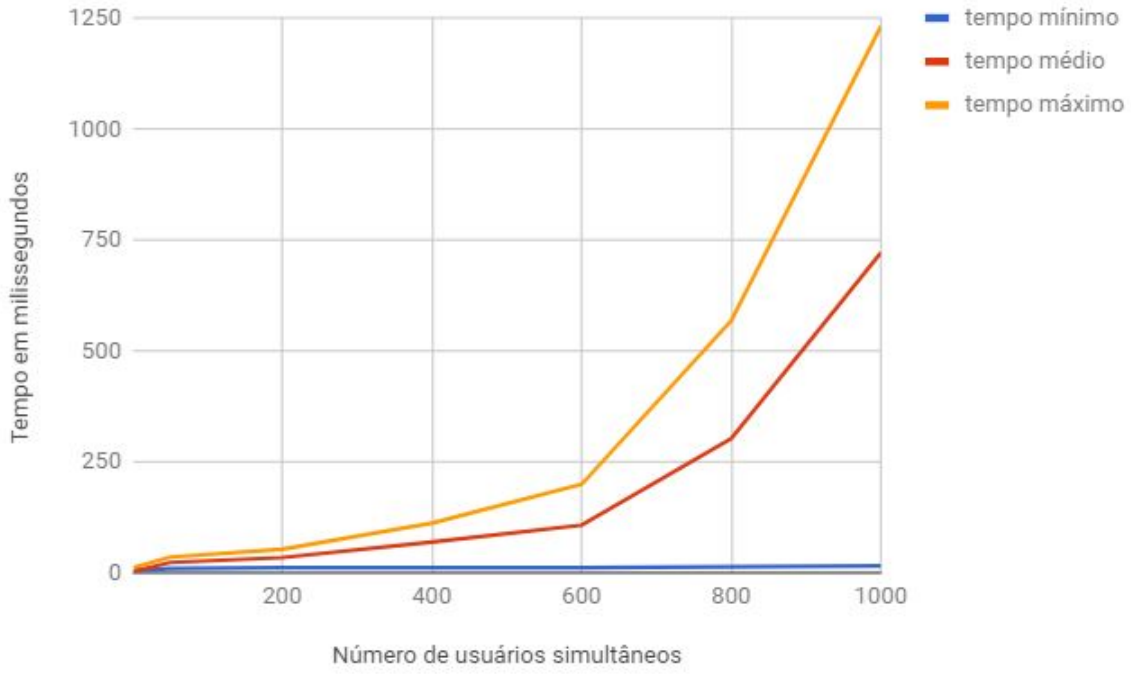
Para os testes de tempo de resposta, a medição foi realizada com uso da plataforma JMeter (FOUNDATION, 2018). Configurando o ambiente de testes, foram realizadas as requisições para uma quantidade de usuários entre 1 e 1000, acessando simultaneamente diversas páginas diferentes do sistema.

Além do teste com quantidade de usuários, os resultados foram calculados com leituras de três máquinas diferentes. A máquina 1, possui um processador i7 4510U até 2.6 GHz (2 núcleos), 16GB RAM DDR3 1600MHz. A máquina 2, possui um processador i3-2100 até 3.1 GHz (2 núcleos), 8GB RAM 1866MHz. A máquina 3, possui um processador Celeron N3060 até 1.6 GHz (2 núcleos), 4GB RAM DDR3 1600MHz.

Além do teste do tempo de resposta, também foi avaliada a quantidade de requisições que retornaram resposta válida ou inválida. O servidor foi alocado primeiramente na máquina 1 e em seguida na máquina 3. Não foram realizados testes com o servidor distribuído paralelamente devido ao tempo para construção do cenário e avaliação dos resultados.

A Figura 30 apresenta o resultado do tempo de resposta para o servidor na máquina 3. A partir dos testes foi possível verificar que o tempo mínimo para a resposta não sofre grande variação, conforme o aumento de carga, mantendo-se em até 20ms. Outra observação é que o tempo para resposta das primeiras 400 requisições permaneceu muito próximo dos tempos para 50 e 200 requisições, mas representando um aumento de até 50% de tempo máximo de 50 para 400 requisições.

Figura 30 – Gráfico de tempo de resposta com sistema na máquina 3.



Fonte: Autoria própria

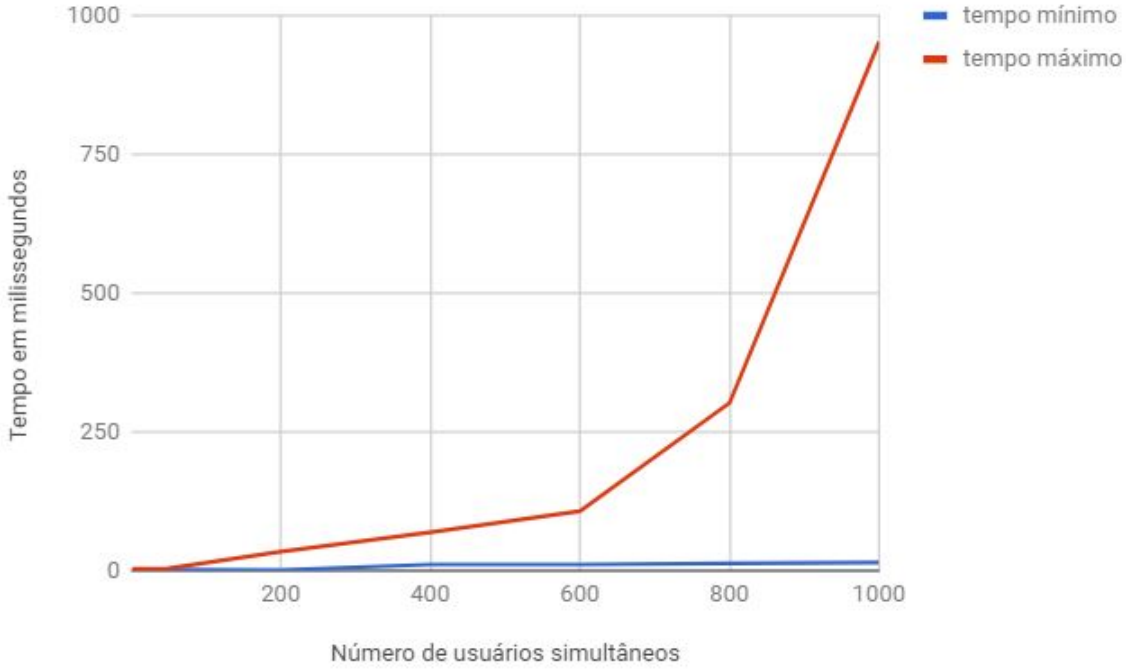
Os valores máximos obtidos demonstram que o sistema é capaz de responder em tempo hábil, sendo o maior valor observado abaixo de 1300ms (1,3 segundos). Essa condição atingida sob o estresse do servidor para responder 1000 requisições paralelas, caso improvável com aplicação do sistema dentro da organização Jocum PG. Contudo, relevante para a aplicação do sistema estendido a organização como todo, ou outras organizações maiores.

Também foi verificado que em quase 100% das requisições o servidor retornou mensagens de sucesso. As duas mensagens com *status* 500, indicativo de erro na requisição, foram apresentadas durante os testes com maior número de requisições simultâneas (1000). No entanto, é importante ressaltar que os erros não ocorreram no mesmo teste, e foram realizados 20 testes diferentes para as 1000 requisições.

Quanto ao tempo de latência demonstrado na Figura 31, foi observado que os valores máximos obtidos estavam próximos ao tempo médio observado na Figura 30. O tempo mínimo variou entre 1ms e 20ms, acompanhando o valor mínimo do tempo de resposta. Para o tempo máximo, foi possível observar os valores mais elevados para os testes com 800 e 1000 requisições, com tempo de latência de 365ms e de aproximadamente 750ms respectivamente.

Tais medidas apontaram um aumento de mais de 100% com o acréscimo de 200 requisições. Essa observação foi um indicativo que a medida que forem adicionadas mais requisições, além de 1000, pode ser viável realizar um *upgrade* de *hardware*, afim de manter o tempo de latência abaixo de 1 segundo, considerando o cenário proposto.

Figura 31 – Gráfico de latência com sistema na máquina 3.



Fonte: Autoria própria

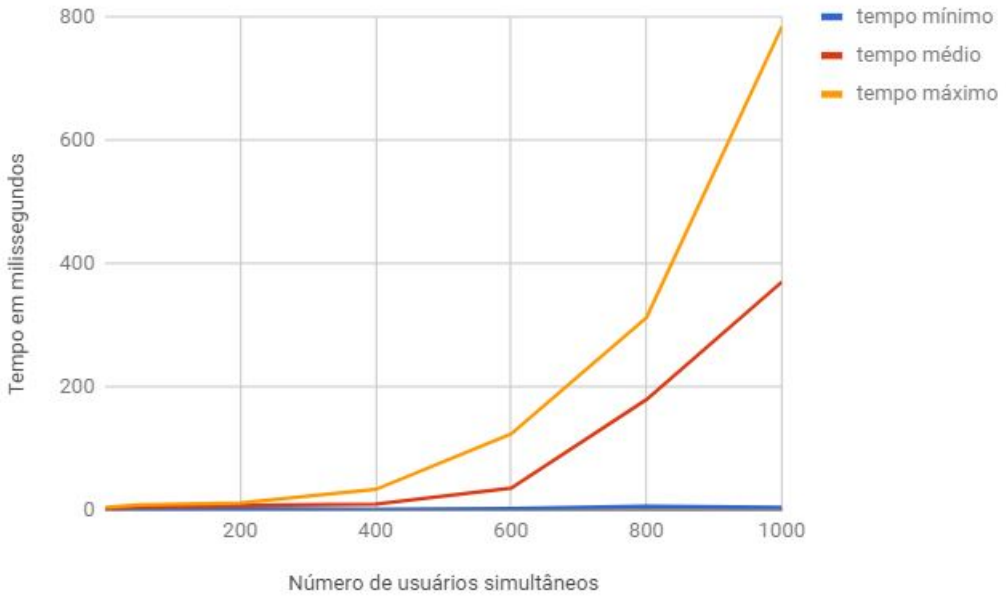
Em todos os testes, realizados a partir dos três pontos da rede, foi possível observar pouca variação dos tempos máximos e mínimos. A máquina com menor tempo, foi a que estava executando o servidor, enquanto as demais apresentaram uma variação, para mais alto, de menos de 10% do tempo dos testes executados no servidor.

Para testar o possível resultado de um aumento de capacidade de *hardware*, o servidor foi transferido para a máquina 1. Desta forma, além de verificar resultado de escalabilidade, pode-se analisar o comportamento do sistema com a variação de ambiente. A Figura 31 mostra o tempo de resposta para os mesmos testes aplicados com o sistema na máquina 3.

É possível notar que todos as medidas de tempo sofreram uma redução. Para o caso de maior carga (1000), a redução do tempo máximo apresentada foi de 37,8%. Foi observado também que em nenhum dos testes realizados, o tempo para qualquer requisição foi superior a 800ms.

Os resultados apontaram que em alguns testes foi possível reduzir metade do tempo máximo gasto na máquina 3. Tais resultados puderam ser observados para 200 e 400 requisições, onde o tempo máximo não ultrapassou a marca de 100ms.

Figura 32 – Gráfico de tempo de resposta com sistema na máquina 1.

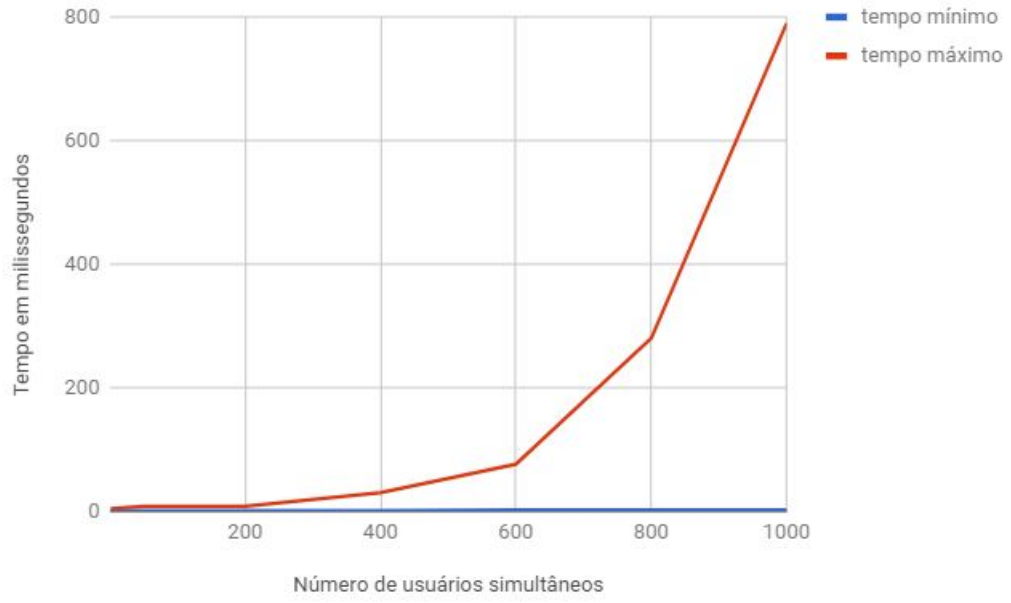


Fonte: Autoria própria

Além das reduções do tempo de resposta, foi possível observar a queda dos tempos de latência, como apresentado na Figura 33. Embora apresente redução, a variação foi menor. Nos testes de 1000 requisições, foi possível observar uma queda de 16% aproximadamente.

Para os demais testes, o tempo máximo de latência teve alterações menores de 10%. Os testes de 600 requisições simultâneas apresentaram um tempo máximo de 166ms contra 92ms e 56ms de 400 e 200 requisições respectivamente.

Figura 33 – Gráfico de latência com sistema na máquina 1.



Fonte: Autoria própria

Todos os testes realizados com o sistema alocado na máquina 1 apresentaram sucesso. Essa medida é um forte indicativo que a capacidade de processamento para o sistema desenvolvido resulta em maior taxa de sucesso quando alocado com recursos maiores.

A análise destes dados junto aos resultados dos gráficos anteriores são um indicativo que não ocorreu sobrecarga do sistema, mesmo que com erro. Com apenas duas mensagens de erro entre mais de 100 testes, é inviável assumir que ocorreu sobrecarga de recursos, mas deve ser levada em consideração sua possibilidade.

Apesar dos indicativos com a escalabilidade, pode-se afirmar que todos os resultados obtiveram valores aceitáveis de funcionamento. Tais resultados demonstram a validade em utilizar o *sistema* completo para o desenvolvimento *web*.

É importante ressaltar que o cenário de testes não demonstra o funcionamento em um ambiente real, no qual o sistema é implantado em um servidor dedicado. Desta forma, os resultados obtidos podem não representar a realidade de funcionamento em uma rede global ou com uso de equipamento dedicado a servidores *web*.

7 CONCLUSÃO

Devido a grande evolução e demanda de recursos na web, as aplicações e arquiteturas precisam ser construídas mais rapidamente. Demandando de soluções práticas que permitam o desenvolvimento e manutenção de forma ágil, além de ter componentes independentes que possam ser alterados ou substituídos.

Conforme apresentado neste trabalho, o uso do Node dedicado ao desenvolvimento do servidor, obedece esses requisitos. Desde sua linguagem, JavaScript, que é amplamente usada, a alta capacidade de oferecer recursos em concorrência e a programação orientada à eventos são características importantes para um servidor. Além destas características, a plataforma Node possui uma ótima documentação e desenvolvedores tem se empenhado para criar módulos e ferramentas de forma a contribuir com a comunidade.

A construção de um servidor usando alguns dos *frameworks*, como Express, permitem o desenvolvimento de uma forma consiste e organizada. A estruturação das rotas, modelos e os meios de comunicação com banco de dados permitem grande flexibilidade e escalabilidade, sem a perda de eficiência.

Deve ser levado em consideração que o Node.Js é uma ferramenta "não optativa", sendo responsabilidade do desenvolvedor definir a organização de código com a qual irá trabalhar. A separação de rotas, modelos e comunicação com o banco, aplicadas a este projeto, permitiu estruturar de forma consistente a solução proposta, que pode ser aplicado em diversos cenários, além do destinado ao atendimento de organizações de assistência social.

Assim como o Node.js, o Angular também conta com uma ampla comunidade, disponibilizando diversos recursos e repositórios. Essas ferramentas são valiosas, uma vez que oferecem ganho de tempo e conhecimento, já que muitas delas contam com exemplos e explicações sobre seu desenvolvimento.

A construção do *template* com Angular permitiu que tarefas comuns e repetitivas, como o desenvolvimento de *views* fosse agilizado, através do uso de várias diretivas e serviços. A construção destes modelos, facilita a configuração e padronização da parte visual do projeto, e também oferece uma solução responsiva de desenvolvimento de *front-end*.

Uma das características importantes alcançadas neste trabalho foi a separação das partes, podendo ser usados separadamente o lado cliente, servidor ou banco. As três ferramentas possuem conexões que podem ser facilmente substituídas por outras, desde que respeitem as regras de conexão e a forma em que os dados estejam trafegando.

7.1 TRABALHOS FUTUROS

O passo seguinte para o projeto é a modelagem dos ciclos adicionais, expandindo a capacidade de atendimento e funcionalidade do sistema. Desta forma, atendendo completamente os requisitos da organização Jocum e outras. Também é interessante construir a comunicação com banco MongoDB, uma vez que soluções entre Node.js e MongoDB tendem a oferecer melhor rendimento a um menor custo.

Por fim, por meio dos resultados obtidos e apresentados no capítulo 6, foi possível concluir e afirmar a viabilidade do uso do sistema Angular, Node.js e SQL Server, através da construção de um sistema *web* completo para uma organização de assistência social. Apesar da complexidade das estruturas, como a comunicação entre servidor e banco ou a definição das diretivas e serviços do Angula, a solução apresentou um resultado consistente e aplicável a opções além das quais foi destinada inicialmente.

Também são considerados abordar e avaliar as limitações deste trabalho. Para isso é sugerido aplicação de testes em um servidor destinado a aplicações *web*, aplicar o sistema na Jocum PG e avaliar a usabilidade do sistema desenvolvido e desenvolver paralelamente recursos da aplicação usando e não usando a base proposta para avaliar o ganho em tempo de desenvolvimento.

REFERÊNCIAS

ABDUKALYKOV, R. *et al.* Quantifying the impact of different non-functional requirements and problem domains on software effort estimation. In: **2011 Ninth International Conference on Software Engineering Research, Management and Applications**. [S.l.: s.n.], 2011. p. 158–165.

ACEMOGLU, D.; ROBINSON, J. A. **Why nations fail: The origins of power, prosperity, and poverty**. [S.l.]: Broadway Business, 2013.

ALEY, R. Managing business logic with functions. In: **Pro Functional PHP Programming**. [S.l.]: Springer, 2017. p. 147–163.

ALI, N.; LAI, R. A method of software requirements specification and validation for global software development. **Requirements Engineering**, Springer, v. 22, n. 2, p. 191–214, 2017.

ANGULARJS. **Date Directive**. 2010. Disponível em: <https://docs.angularjs.org/api/ng/filter/date>. Acesso em: 27 de Abril de 2018.

ANWAR, N. Architecting scalable web application with scalable cloud platform. Metropolia Ammattikorkeakoulu, 2018.

BELLARD, F. **JSLinux**. 2017. Disponível em: <https://bellard.org/jslinux/>. Acesso em: 28 de Abril de 2018.

BEZERRA, E. **Princípios de Análise e Projeto de Sistema com UML**. [S.l.]: Elsevier Brasil, 2017. v. 3.

BOOTSTRAP. **Typeahead**. 2016. Disponível em: <https://ng-bootstrap.github.io/#/components/typeahead/>. Acesso em: 27 de Abril de 2018.

CANTELON, M. *et al.* **Node. js in Action**. [S.l.]: Manning Publications, 2017.

CECHINEL, A. *et al.* Avaliação do framework angular e das bibliotecas react e knockout para o desenvolvimento do frontend de aplicações web. Florianópolis, SC, 2017.

CHIEU, T. C.; MOHINDRA, A.; KARVE, A. A. Scalability and performance of web applications in a compute cloud. In: IEEE. **e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on**. [S.l.], 2011. p. 317–323.

CHODOROW, K. **MongoDB: The Definitive Guide: Powerful and Scalable Data Storage**. [S.l.]: "O'Reilly Media, Inc.", 2013.

CHOWDHURY, F.; DESAI, S.; AUDRETSCH, D. B. **Corruption, Entrepreneurship, and Social Welfare: A Global Perspective**. [S.l.]: Springer, 2017.

CORDEIRO, D. d. A. **Estudo de escalabilidade de servidores baseados em eventos em sitemas multiprocessados: um estudo de caso completo**. 2006. Tese (Doutorado) — Universidade de São Paulo, 2006.

COULOURIS, G. *et al.* **Sistemas Distribuídos-: Conceitos e Projeto**. [S.l.]: Bookman Editora, 2013.

COUTO, B. R. *et al.* **O Sistema Único de Assistência Social no Brasil: uma realidade em movimento**. [S.l.]: Cortez Editora, 2014.

DABEK, F. *et al.* Event-driven programming for robust software. In: ACM. **Proceedings of the 10th workshop on ACM SIGOPS European workshop**. [S.l.], 2002. p. 186–189.

DOUGLAS, M.; MARABESI, M. **Aprendendo Laravel: O framework PHP dos artesãos da web**. [S.l.]: Novatec Editora, 2017.

ELMASRI, R. *et al.* Sistemas de banco de dados. Pearson Addison Wesley, 2005.

FEDOSEJEV, A. **React. js Essentials**. [S.l.]: Packt Publishing Ltd, 2015.

FOUNDATION, T. A. S. **Apache JMeter**. 2018. Disponível em: <https://jmeter.apache.org/>. Acesso em: 27 de Abril de 2018.

GOODYEAR, M. **Enterprise System Architectures: Building Client Server and Web Based Systems**. [S.l.]: CRC press, 2017.

HOTFRAMEWORKS. **Find your new favorite web framework**. 2018. Disponível em: <https://hotframeworks.com/>. Acesso em: 03 de Março de 2018.

IBGE. **As entidades de assistência social privadas sem fins lucrativos no Brasil, 2014-2015: unidades de prestação de serviços socioassistenciais**. Rio de Janeiro: Instituto Brasileiro de Geografia e Estatística - IBGE, 2015. ISBN 978-85-240-4360-4.

JAVATPOINT. **Understanding Synchronous vs Asynchronous**. 2015. Disponível em: <https://www.javatpoint.com/understanding-synchronous-vs-asynchronous>. Acesso em: 28 de Abril de 2018.

JOCUM. **Apresentação oficial da Jocum em Ponta Grossa**. [S.l.], 2014. 9 p.

KERZNER, H.; KERZNER, H. R. **Project management: a systems approach to planning, scheduling, and controlling**. [S.l.]: John Wiley & Sons, 2017.

LAUDON, K. C.; LAUDON, J. P. **Management information system**. [S.l.]: Pearson Education India, 2016.

LEE, C.; CLERKIN, R. M. The adoption of outcome measurement in human service nonprofits. **Journal of Public and Nonprofit Affairs**, v. 3, n. 2, p. 111–134, 2017.

LEE, R. L.; BLOUIN, M. C. Exploring the factors associated with online financial and performance disclosure in nonprofits. **Journal of the Southern Association for Information Systems**, Michigan Publishing, University of Michigan Library, v. 3, n. 1, 2015.

LIU, H. H. **Software performance and scalability: a quantitative approach**. [S.l.]: John Wiley & Sons, 2011. v. 7.

MACHADO, F. N. R. Análise e gestão de requisitos de software: onde nascem os sistemas. **1ª edição, Editora Érica, São Paulo**, 2011.

NEWMAN, S. **Building microservices: designing fine-grained systems**. [S.l.]: "O'Reilly Media, Inc.", 2015.

OZGUR, C. *et al.* A comparative study of network modeling using a relational database (ie oracle, mysql, sqlserver hadoop) vs. neo4j. In: **2017 Annual Meeting of Midwest Decision Sciences Institute Meeting,(MWDSI) April**. [S.l.: s.n.], 2017. p. 156–165.

PEREIRA, C. R. **Aplicações web real-time com Node. js**. [S.l.]: Editora Casa do Código, 2014.

POPESCU, A. **Nosql at codemash—an interesting nosql categorization**. 2010.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software-8ª Edição**. [S.l.]: McGraw Hill Brasil, 2016.

REACT. **React**. 2018. Disponível em: <https://reactjs.org/>. Acesso em: 28 de Abril de 2018.

REACTPHP. **ReactPHP**. 2017. Disponível em: <https://reactphp.org/>. Acesso em: 28 de Abril de 2018.

ROSATO, F. **Vantagens e desvantagens de uma arquitetura microservices**. 2015. Disponível em: <https://pt.slideshare.net/frosato/vantagens-e-desvantagens-de-uma-arquitetura-microservices>. Acesso em: 28 de Abril de 2018.

SAUVE, J. P. **Introdução e Motivação: Arquiteturas em n Camadas**. 2016. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/j2ee/html/intro/intro.htm>. Acesso em: 28 de Abril de 2018.

SAVCHENKO, D. I.; RADCHENKO, G. I.; TAIPALE, O. Microservices validation: Mjolnirr platform case study. In: IEEE. **Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on**. [S.l.], 2015. p. 235–240.

SCHIECK, R. **Threads: paralelizando tarefas com os diferentes recursos do Java**. 2017. Disponível em: <https://www.devmedia.com.br/threads-paralelizando-tarefas-com-os-diferentes-recursos-do-java/34309>. Acesso em: 28 de Abril de 2018.

SESHADRI, S.; GREEN, B. Desenvolvendo com angularjs. **Novatec Editora, Sao Paulo, SP**, 2014.

SILBERSCHATZ, A.; KORTH, H.; SUNDARSHAN, S. **Sistema de banco de dados**. [S.l.]: Elsevier Brasil, 2016.

SKVORC, B. Best php frameworks for 2014. **Sitepoint**, 2013. Disponível em: <http://www.sitepoint.com/best-php-frameworks-2014/>. Acesso em: 28 de Abril de 2018.

SOMMERVILLE, I. Arquitetura orientada a serviços. **Engenharia de Software**, p. 355–368, 2011.

STUBBS, J.; MOREIRA, W.; DOOLEY, R. Distributed systems of microservices using docker and serfnode. In: IEEE. **Science Gateways (IWSG), 2015 7th International Workshop on**. [S.l.], 2015. p. 34–39.

TEIXEIRA, P. **Professional Node. js: Building Javascript based scalable software**. [S.l.]: John Wiley & Sons, 2012.

TILKOV, S.; VINOSKI, S. Node. js: Using javascript to build high-performance network programs. **IEEE Internet Computing**, IEEE, v. 14, n. 6, p. 80–83, 2010.

TIWARI, S. **Professional NoSQL**. [S.l.]: John Wiley & Sons, 2011.

TRIERVEILER, H. J.; SELL, D.; PACHECO, R. C. dos S. A importância do conhecimento organizacional para o processo de inovação no modelo de negócio. **Navus-Revista de Gestão e Tecnologia**, Serviço Nacional de Aprendizagem Comercial, v. 5, n. 1, p. 113–126, 2015.

VLADUTU, A. **Mastering Web Application Development with Express**. [S.l.]: Packt Publishing Ltd, 2014.

WILSON, J. **Node.js 8 the Right Way: Practical, Server-side Javascript that Scales**. [S.l.]: Pragmatic Bookshelf, 2018.