Avaliação e Evolução Arquitetural de um Sistema Web ERP

Igor Leal Antunes

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2017

Igor Leal Antunes

Avaliação e Evolução Arquitetural de um Sistema Web ERP

Monografia apresentada ao curso Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Raoni Kulesza

Dezembro de 2017

Ficha Catalográfica elaborada por
Rogério Ferreira Marques CRB15/690

A627a Antunes, Igor Leal.
Avaliação e evolução arquitetural de um sistema Web ERP / Igor Leal Antunes. – João Pessoa, 2017.
91p. : il.

Monografia (Bacharelado em Ciência da Computação) – Universidade Federal da Paraíba - UFPB.
Orientador: Profº. Dr. Raoni Kulesza.

1. Informática. 2. Sistemas ERP. 3. Arquitetura de software. 4. Sistemas Web. I. Título.

UFPB/BSCI CDU: 004. (043.2)

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado **Avaliação e Evolução Arquitetural de um Sistema Web ERP** de autoria de **Igor Leal Antunes**, aprovada pela banca examinadora constituída pelos seguintes professores:

____
Prof. Dr. Raoni Kulesza
CI/UFPB

____
Prof[a]. Dra. Ayla Débora Dantas de Souza Rebouças
CCAE/UFPB

____
Prof. MsC. Carlos Eduardo Dias Silveira
CCAE/UFPB

____
Coordenador do Departamento de Informática
Prof. Dr. Gustavo Henrique Matos Bezerra Motta
CI/UFPB

João Pessoa, 04 de dezembro de 2017.

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

"As pessoas costumam dizer que a motivação não dura para sempre. Bem, nem o efeito do banho, por isso recomenda-se diariamente." (Zig Ziglar)

DEDICATÓRIA

Dedico este trabalho a minha família e a todos os meus professores.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais e ao meu irmão por sempre me apoiarem e me darem forças nas minhas decisões, permitindo que eu alcançasse meus objetivos na minha vida. Sem eles seria impossível chegar até aqui.

Também agradeço a minha namorada e companheira Ana Beatriz por me ajudar na correção e formatação deste trabalho; e principalmente por me incentivar nesse processo cansativo de produção. Agradeço por todos os beijos e carinho.

Agradeço aos meus colegas mais próximos de curso: Igor Nóbrega, Thais Ramos, Karla Tabosa, Fernando Henrique, Paulo Branco e Gleydson Morais por terem me fornecido grandes experiências dentro e fora do curso de computação.

Agradeço a todos os professores que fizeram parte da minha formação, pois cada um deles contribuiu para a pessoa que me tornei de forma pessoal e academicamente. Agradeço especialmente ao meu orientador Prof. Dr. Raoni Kulesza pelo esforço e bom trabalho durante a produção deste trabalho.

Gostaria de agradecer ao grupo PET.com por ter me proporcionado grandes experiências de pesquisa ensino e extensão e também por ter me proporcionado grandes amizades.

RESUMO

Sistemas de Informação se mostram cada vez mais indispensáveis nas mais diversas corporações. Sem eles, grande parte das empresas não conseguiriam realizar suas operações e tarefas de uma maneira eficiente. Os sistemas passaram por diversas transformações ao longo dos anos, principalmente com o surgimento da *Web*, ocasionando uma mudança drástica nesse tipo de sistema. Todos os dias novas arquiteturas, tecnologias e ferramentas surgem para que esses sistemas se tornem cada vez mais poderosos e robustos, trazendo mais benefícios para as empresas. Este trabalho tem como principal objetivo a avaliação de um sistema Web ERP que utiliza um modelo arquitetural MVC implementado com um framework PHP. Para tanto, foi realizada uma revisão bibliográfica acerca de sistemas Web, uma especificação de requisitos e projeto do sistema avaliado. Adicionalmente, também são apresentadas propostas de evolução arquitetural do sistema, bem com uma discussão de resultados de uma avaliação inicial das mudanças. Por fim, são discutidas as considerações finais e propostas de trabalhos futuros.

Palavras-chave: Sistemas ERP, Sistemas Web, PHP, Codeigniter, frameworks Web, Arquitetura de Software.

ABSTRACT

Information systems are becoming more and more indispensable in the most diverse corporations. Without them, most companies would not be able to perform their operations and tasks efficiently. The systems underwent several transformations over the years, mainly with the appearance of the Web, causing a drastic change in this type of system. Every day new architectures, technologies and tools emerge so that these systems become increasingly powerful and robust, bringing more benefits to companies. This work has as main objective the evaluation of an ERP Web system that uses an MVC architectural model implemented with a PHP framework. For this, a bibliographic review was performed on Web based systems, a specification of requirements and design of the evaluated system. In addition, proposals are also presented for the architectural evolution of the system, as well as a discussion of the results of an initial evaluation of the changes. Finally, the final considerations and proposals for future work are discussed.

Keywords: Software Requirements Specification, Software Architecture, Web Systems, ERP Systems, Software Evaluation.

LISTA DE FIGURAS

Figura 1: Sistemas OLAP e OLTP 15
Figura 2: Transferência estática de arquivos 20
Figura 3: Forma primitiva de gerar conteúdo de forma dinâmica 21
Figura 4: Servidor utilizando um programa CGI para gerar conteúdos dinâmicos 21
Figura 5: Comparação de uso entre programas CGI e *template systems* 23
Figura 6:Modelo tradicional vs AJAX 24
Figura 7: Arquitetura Cliente-Servidor de duas camadas 26
Figura 8: Exemplo de Arquitetura Cliente-Servidor de Múltiplas-camadas 27
Figura 9: Exemplo do desacoplamento gerado pela arquitetura SOA 28
Figura 10: Diferentes tecnologias na arquitetura de Micro-serviços 33
Figura 11: exemplo de escalabilidade específica no Micro-serviços 33
Figura 12: Modelo MVC original 36
Figura 13: Modelo 2 do MVC 37
Figura 14: Arquitetura MVP 38
Figura 15: Arquitetura MVVM 39
Figura 16: Ciclo de vida do CodeIgniter 41
Figura 17: Interação de frameworks *front-end e back-end* 42
Figura 18: Comparativo entre os principais paradigmas de banco de dados 49
Figura 19: Diagrama UML de caso de uso 58
Figura 20: Arquitetura de alto-nível do sistema 69
Figura 21: Arquitetura de baixo nível 72
Figura 22: Comparação entre sistemas tradicionais e SPA 74
Figura 23: Fluxo de atividade (BPMN) 76
Figura 24: Fluxo de chamadas 77
Figura 25: Fluxo detalhado de chamadas 77
Figura 26: Fluxo de chamadas com framework SPA 79
Figura 27: Comparação de desempenho médio das arquiteturas SPA e Tradicional 81
Figura 28: Comparação de desempenho geral das arquiteturas SPA e Tradicional 81
Figura 29: Arquitetura de Micro-Serviços 83
Figura 30: Arquitetura de Híbrida de Micro-Serviços 84
Figura 31: Arquitetura de Híbrida de Micro-Serviços com chamadas diretas pelos clientes 85

LISTA DE ABREVIATURAS

AJAX	–	Asynchronous JavaScript and XML
API	–	Application Programming Interface
HTML	–	HyperText Markup Language
HTTP	–	HyperText Transfer Protocol
JSON	–	JavaScript Object Notation
JSP	–	JavaServer Pages
MVC	–	Model-View-Controller
SQL	–	Structured Query Language
CGI	–	Common Gateway Interface
JVM	–	Java Virtual Machine
SOAP	–	Simple Object Access Protocol
WSDL	–	Web Service Definition Language
UDDI	–	Universal Description, Discovery and Integration
AWS	–	Amazon Web Services
PHP	–	Hypertext Preprocessor
OLTP	–	Online transaction processing
OLAP	–	Online analytical processing
ERP	–	Enterprise Resource Planning
POS	–	Point of Sales

1. Introdução 15
1.1 Objetivos gerais 18
1.2 Objetivos Específicos 18
2. Fundamentação Teórica 19
2.1. Evolução da Internet 20
2.1.1 Common Gateway Interface (CGI) 21
2.1.2 FastCGI 22
2.1.3 AJAX 23
2.2. Arquitetura de sistemas 25
2.2.1 Arquitetura Cliente-Servidor 25
2.2.1.1 Cliente-Servidor de duas camadas 25
2.2.1.2 Cliente-Servidor múltiplas camadas 26
2.2.2 Arquitetura orientada a serviços (*SOA*) 27
2.2.2.1 Princípio fundamental da arquitetura SOA 28
2.2.2 Protocolos de comunicação 28
2.2.2.3 Níveis de maturidade do *REST* 30
2.2.2.4 Popularidade do *REST* 31
2.2. 3. Arquitetura de Micro-serviços 31
3. Arquitetura MVC 35
3.1 Proposta Original do MVC 35
3.2 Modelo 2 do MVC 36
3.3 Arquitetura *MVP* 37
3.4. Arquitetura MVVM 38
4. Frameworks Web 40
4.1 CodeIgniter 40
4.2 KnockoutJS 41
5. *Single Page Web-Application* 43
6. PHP 44
7. Banco de Dados 44
7.1 Bancos Relacionais 45
7.2 Bancos Orientados a Objeto 46
7.3 Bancos *NoSQL* 46
7.4 Bancos *NewSQL* 47
3. O sistema 50
3.1 Visão Geral 50
3.2 Requisitos Funcionais 51
[RF01] Receber submissão de pedidos do sistema POS 51
[RF02] Atualização de medidas dos clientes 51
[RF03] Atualização de design do produto 51
[RF04] Atualização do padrão corporal de cada cliente 51
[RF05] Geração de PDF para produção 52
[RF06] Organização de logística 52
[RF07] Agendamento de provas 52
[RF08] Processo de correção 52
[RF09] Recebimento de pagamentos 53

[RF10] Finalizar pedido 53
[RF11] Tradução de PDFs 53
[RF12] Atualização da forma corporal de cada cliente 53
[RF13] Notificação de erros de produção 54
3.3 Requisitos Não Funcionais 54
Sistema 54
[RNF01] Especificação de Software 54
[RNF02] Banco de Dados 54
[RNF03] Servidor 54
[RNF04] Versão da biblioteca *Jquery UI* 55
[RNF05] Versão da biblioteca *Bootstrap* 55
RNF06] Versão da biblioteca *Raphael.js* 55
RNF07] Versão da biblioteca *Tagdd.js* 55
RNF08] Versão da biblioteca FFPDF 55
RNF08] Versão da biblioteca FFMPEG 55
RNF09] Versão da biblioteca *GroceryCrud* 56
Segurança 56
[RNF10] Recebimento de dados via *End-point* 56
[RNF11] Comunicação segura entre cliente e servidor 56
[RNF12] Backup de Dados 56
[RNF13] Dados sensíveis 56
[RNF14] Comunicação entre sistemas 57
Portabilidade 57
[RNF15] Dispositivos suportados 57
Internacionalização 57
[RNF16] Idioma 57
3.4 Casos de uso 58
3.4.1 [UC01] – Submissão de venda 59
3.4.2 [UC02] – Processar Pagamentos 59
3.4.3 [UC03] – Agendamento de prova 60
3.4.4 [UC04] – Atualizar medidas 60
3.4.5 [UC05] – Atualizar Design 61
3.4.6 [UC06] – Atualizar padrão corporal 61
3.4.7 [UC07] – Organização de logística 62
3.4.8 [UC08] – Geração de PDF para produção 63
3.4.9 [UC09] – Tradução de PDF de produção 64
3.4.10 [UC10] – Notificar erros de produção 64
3.4.11 [UC11] – Processo de correção de roupas 65
3.4.12 [UC12] – Atualizar forma corporal 65
3.4.13 [UC13] – Finalizar pedido 66
3.5 Arquitetura 67
3.5.1 Arquitetura de baixo Nível 70
4. Avaliação e proposta de evolução da arquitetura 73
4.1 Contexto 73
4.1 Visão geral do processo de negócio 74
4.2 Visão detalhada do processo de negócio – Camada de Visão 76
4.3 Visão detalhada do processo de negócio – Camada de Negócio 78

4.4 Novo fluxo de execução ... 78
4.5 Comparação de desempenho ... 79
4.6 Análise de evolução para Micro-serviço ... 82

5. Considerações Finais e Trabalhos Futuros ... 86

1. Introdução

Grande parte das organizações empresariais de hoje utilizam algum tipo de Sistema de Informação para auxiliar nos seus processos. Muitas vezes diversos processos estão envolvidos nas atividades das empresas e demandam um complexo gerenciamento de informação, fazendo-se indispensável o uso desses sistemas. Os Sistemas de Informação também são indispensáveis para agilizar atividades da empresa e, consequentemente, proporcionar maior retorno financeiro.

Sistemas *OLAP* (do inglês, *On-Line Analytical Processing*) e *OLTP* (do inglês, *On-Line Transaction Processing*) vêm sendo utilizados pelas empresas para realizar suas atividades. Cada tipo de sistema possui características e objetivos distintos (ver figura 1).

Os sistemas *OLAP* são focados em realizar análises sobre dados de uma organização a fim de ajudar no processo de tomada de decisão. Esses sistemas trabalham muito próximos ao banco de dados das organizações e buscam organizar e extrair informações relevantes sobre os processos da empresa, sendo muito empregados também em processos de mineração de dados [48].

Já os sistemas *OLTP* são responsáveis pelas atividades mais rotineiras de uma empresa, tais como: realizações de compras, atualização de inventário, processamento de pagamentos, entre outros [48]. Os sistemas *OLTP* devem ter um bom tempo de resposta, uma vez que podem ser utilizados por vários usuários ao mesmo tempo e não devem atrasar os processos realizados no dia a dia das empresas.

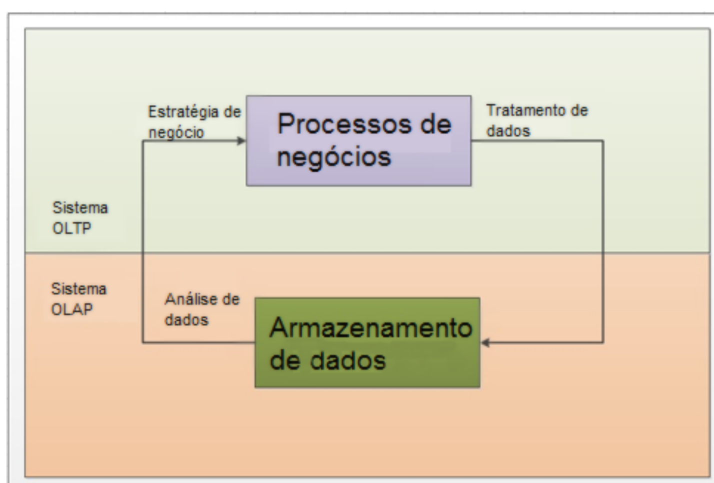

Figura 1: Sistemas OLAP e OLTP
Fonte: Introduction on Data Warehouse with OLTP and OLAP, 2013 p. 2572.

O sistema em estudo é classificado com um sistema OLTP, mais especificamente, um sistema ERP (do inglês, *Enterprise Resource Planning*) dado que ele é voltado para lidar com as atividades rotineiras da empresa. A empresa trabalha no ramo de roupas de luxo sob medida e tem pretensão de expandir seus negócios para outros países, tornando crucial o uso de um sistema de informação robusto. Esse ramo de negócio possui diversos processos, alguns mais genéricos e outros bastante específicos desse ramo, possuindo processos de venda, acompanhamento, edição do produto; notificação de usuários, processos de prova, processos de alteração e de logística.

A arquitetura de sistemas *Web* evoluiu drasticamente desde a criação da Internet. No início os sistemas eram feitos utilizando a arquitetura CGI (do inglês, *Common Gateway Interface*). Essa interface deu um grande poder aos servidores, já que passou a oferecer a capacidade de executar *scripts* de códigos – geralmente *Perl* – ao processar as requisições HTTP [1], fazendo com que os sistemas Web pudessem processar requisições de uma forma mais dinâmica. Por motivos de desempenho, os servidores tiveram que evoluir a forma de processar essas requisições, surgiram então os módulos e o *FastCGI* (versão aprimorada do *CGI* clássico). Essa tecnologia permitiu que os servidores suportassem mais requisições e usuários [2].

Outro problema também emergiu no início da *Web:* era muito difícil separar os códigos de apresentação e lógica das aplicações, dificultando e tornando lento o seu desenvolvimento. Surgiram então os *template systems,* eles permitiam que códigos executáveis de uma linguagem de programação fossem inseridos diretamente nos arquivos responsáveis pela apresentação do sistema; dessa forma, a arquitetura ganhou uma melhor separação entre as camadas de lógica e de apresentação [1]. Depois disso, diversas arquiteturas surgiram, entre elas o "modelo 2" da arquitetura MVC, que mais tarde tornou-se um dos principais modelos de sistemas *Web* [4].

Surgiram também protocolos de comunicação (*SOAP, REST, etc*) que permitiam que os sistemas se comunicassem independentemente da linguagem de programação utilizada. Com isso, os desenvolvedores não estavam mais somente construindo aplicações que serviam conteúdos para os navegadores; mas sim, sistemas complexos que envolviam várias camadas de comunicação interna e externa (com outros sistemas) [3]. A partir daí os sistemas cresceram bastante, e a quantidade de usuários aumentou consideravelmente, fazendo com que esses sistemas se tornassem grandes demais, transformando-os em gigantes sistemas Monolíticos [5]. Esses sistemas possuem diversos problemas de escalabilidade e desempenho

quando muitos usuários o utilizam. A solução foi encontrada em arquiteturas menos monolíticas e mais distribuídas – como as de *SOA (*do inglês, *service-oriented- architecture)* e Micro-serviços. Esses modelos possuem uma melhor distribuição de cada serviço do sistema, fazendo com que a carga de requisições em cada um seja melhor distribuída, melhorando consideravelmente a escalabilidade e estabilidade do sistema [5].

Esse trabalho tem como objetivos gerais de

em que os sistemas web tenham uma velocidade e usabilidade comparáveis com sistemas desktop tradicionais [47]. Esses frameworks utilizam arquiteturas conhecidas como SPA (do inglês, *single page application*) e permitem que os sistemas *Web* tenham uma usabilidade similar aos sistemas desktop tradicionais, eliminando a necessidade de recarregar a página frequentemente [47], atualizando somente o necessário através do uso de Javascript e comunicações *AJAX* com o servidor [47]. Esse modelo remove a responsabilidade de gerar a camada de visão dos servidores, deixando os sistemas mais leves, rápidos e fluídos [47].

A arquitetura do sistema não está uniforme ao longo das suas seções, e com isso, algumas partes do sistema apresentam características muito primitivas e não mais utilizadas. Algumas funcionalidades do sistema, desperdiçam muito tempo e tornam a usabilidade pouco agradável, visto que o sistema é grande e algumas operações podem demorar muito a serem concluídas. Diversos fatores estão relacionados com isso, sendo o principal a arquitetura atual do sistema, mais especificamente, requisitos não-funcionais relacionados ao desempenho da execução de atividades do sistema:

1. **Tempo de execução de processos:** usuários do sistema perdem muito tempo quando realizam tarefas dentro do sistema, ocasionando em uma má utilização dos recursos da empresa.
2. **Usabilidade:** sistemas lentos causam grande desconforto por parte dos seus usuários, deixando muito a desejar em termos de usabilidade.
3. **Capacidade:** em épocas de pico, a empresa atinge seu potencial máximo de clientes rapidamente, chegando a recusar novas vendas por falta de tempo para que se processem os produtos.

O sistema possui hoje uma arquitetura tradicional e é implementado utilizando um *framework* MVC e tecnologias *Web*. O Sistema possui uma navegação tradicional, na qual, as

páginas utilizadas na camada de visão são geradas pelo servidor e então enviadas pelos clientes, fazendo com que tarefas simples possam levar muito tempo para serem concluídas. Algumas seções utilizam requisições AJAX para evitar o recarregamento total da página e melhorar a velocidade e usabilidade do sistema, embora o problema de recarregamento ainda continue presente em grande parte do sistema, visto que o framework atual não permite que o sistema utilize o paradigma de SPA na sua camada de apresentação.

1.1 Objetivos gerais

Esse trabalho tem como objetivos gerais descrever o sistema em questão por meio de engenharia de requisitos e também sua arquitetura. Também, avaliar e propor melhorias na arquitetura atual do sistema.

1.2 Objetivos Específicos

Como objetivos específicos, pretende-se:

1. Revisão da literatura sobre sistemas Web;
2. Realizar a especificação de requisitos e projeto de software do sistema ERP;
3. Propor e avaliar melhorias na arquitetura da camada de visão do sistema por meio de um framework que utilize o modelo de sistemas SPA; e
4. Propor melhorias na arquitetura das camadas mais internas no sistema para uma arquitetura baseada em SOA (Micro-serviços).

2. Fundamentação Teórica

Os sistemas de informação passaram por grandes mudanças ao longo dos anos, evoluindo sua arquitetura, modelos de comunicação, banco de dados e frameworks de aplicação. A chegada da internet foi um dos marcos para essa evolução. A mudança fez com que os sistemas se tornassem bem mais distribuídos, uma vez que quase toda a comunicação nos sistemas é feita através da rede. A transição para a internet fez com que os sistemas migrassem do paradigma *stand-alone* para o distribuído. Com isso, fizeram-se necessárias novas tecnologias, entre elas o surgimento do *fastCGI, Flash* e *AJAX.*

Os modelos distribuídos de sistema também tiveram um papel essencial para os sistemas de informação, podendo ser utilizados nas mais diversas arquiteturas, desde as mais simples arquiteturas cliente-servidor até as mais complexas arquiteturas baseadas em serviços. Hoje, sistemas de grande porte utilizam técnicas elaboradas para suprir a sua grande demanda e prover uma boa experiência para seus usuários, tal como o *Netflix* com seu modelo baseado em Micro-serviços.

As arquiteturas baseadas em *MVC (*do inglês, *Model-View-Controller)*, também sofreram uma grande evolução, novas variantes dessa arquitetura foram criadas e hoje temos uma grande quantidade de Frameworks de aplicação para facilitar e simplificar a construção dos sistemas.

Uma das mais importantes partes dos sistemas de informação é o banco de dados. Eles concentram quase todo o conhecimento e dados dos sistemas, desde simples cadastros de usuários até complexas regras de negócio. Os sistemas passaram a utilizar e produzir cada vez mais informação. Com isso, novos paradigmas de banco de dados foram criados para suprir essa grande demanda. Dessa forma, temos hoje os mais variados paradigmas de bancos, cada um especializado em um tipo de sistema, dos quais podemos escolher o mais adequado para cada tipo de sistema.

2.1. Evolução da Internet

A computação evoluiu a um ritmo muito elevado nas últimas décadas. O hardware ganhou muito poder e o custo de produção caiu, com isso, os softwares também sofreram mudanças drásticas na sua forma (Gabriel Banfalvi, 2012). Com toda essa evolução, novas arquiteturas e métodos tiveram que ser desenvolvidos para apoiá-la.

A primeira versão da Internet originou-se durante o início da década de 1990. Desde aquela época, inúmeras mudanças aconteceram, fazendo com que a ela se tornasse muito maior do que previsto pelo seu criador Tim Berners-Lee. (BERNERS-LEE, 1996).

Nos seus primeiros anos, os servidores trabalhavam de uma maneira bastante simplificada, como representado na figura 2. O cliente requisitava algum recurso contido no servidor através de URLs (do inglês, *Uniform Resource Locator*) e o servidor criava uma resposta que seria enviada ao cliente; esse modelo era chamado de *Read Only Web (BENIOFF, 2008).* Toda a comunicação cliente-servidor era efetuada dessa forma, arquivos estáticos contidos no servidor eram enviados aos clientes. Naquela época, havia pouca necessidade de se gerar conteúdos dinâmicos, pois a *web* possuía poucos usuários e raríssimos sistemas de informação, comparando-se ao que temos hoje (DUANE, 2012).

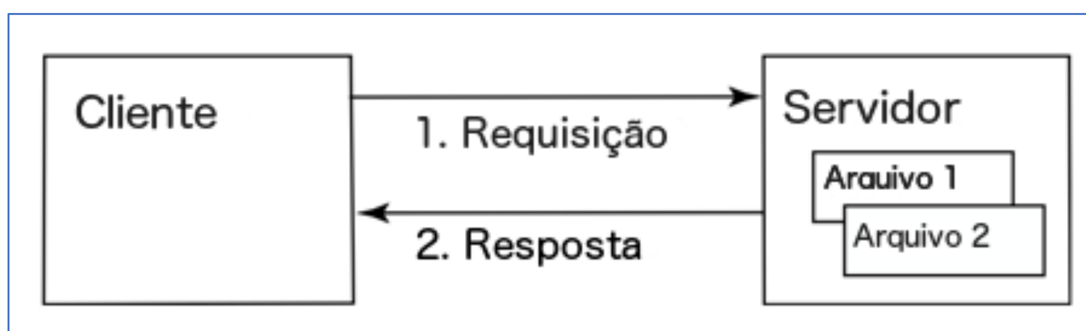

Figura 2: Transferência estática de arquivos
Fonte: Web development with JSP, 2011, p. 3.

Segundo Duane (2012), a crescente popularização da internet criou a necessidade de gerar páginas com conteúdos um pouco mais dinâmicos. Essa necessidade deu origem a uma técnica primitiva de atualização de páginas: um programa rodando em background no servidor atualiza o conteúdo de recursos constantemente (geralmente arquivos HTML), alterando seus conteúdos a partir de uma fonte externa de informação. Essa técnica mostrou-se muito útil em serviços que necessitavam de um certo dinamismo nas suas páginas, como

por exemplo portais de notícias ou previsões de tempo. Assim, o servidor tinha uma forma primitiva de gerar páginas dinâmicas.

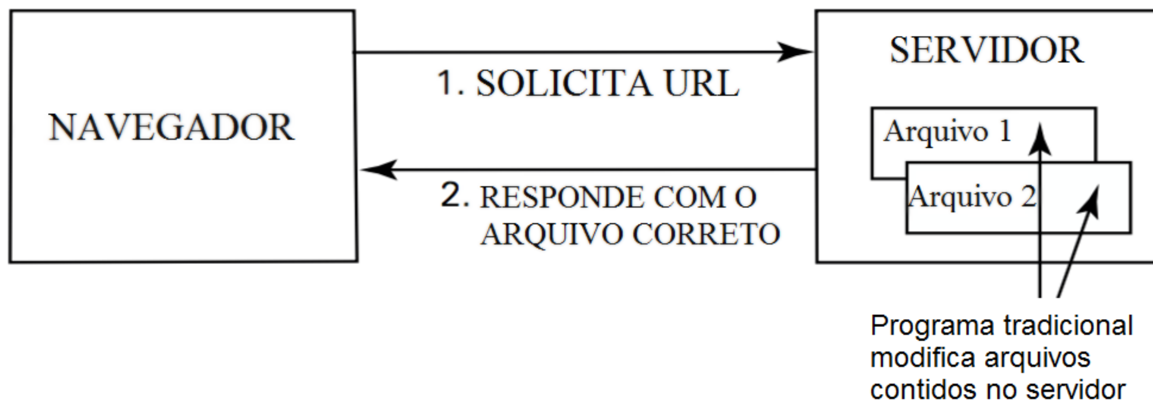

Figura 3: Forma primitiva de gerar conteúdo de forma dinâmica
Fonte: Web development with JSP, 2011, p. 3.

Apesar dessa evolução, o servidor ainda não era capaz de responder dinamicamente a requisições específicas de cada cliente. Então, fez-se necessária alguma técnica que fosse capaz de ler entradas fornecidas pelos clientes, processar essas entradas e, a partir daí, gerar páginas dinâmicas.

2.1.1 Common Gateway Interface (CGI)

CGI (do inglês, *Common Gateway Interface*) define uma interface para que servidores possam se comunicar com programas externos e redirecionar sua saída para o cliente, oferecendo a capacidade de processar dados vindos dos clientes através de programas localizados fora dos servidores (DUANE, 2012). O fluxo de execução de um servidor utilizando *CGI* pode ser representado na seguinte figura:

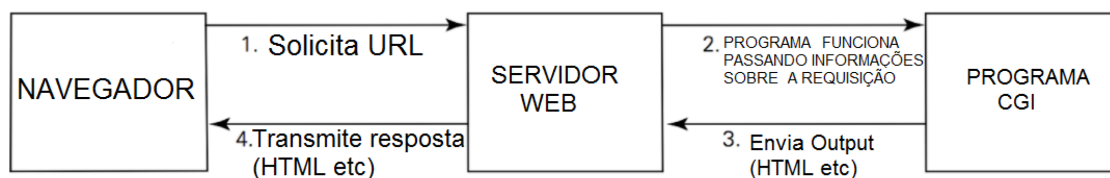

Figura 4: Servidor utilizando um programa CGI para gerar conteúdos dinâmicos
Fonte: Web development with JSP, 2011, p. 5.

O CGI não é uma *API* voltada para uma linguagem de programação específica [1]. Na prática, ele permite que os servidores executem programas das mais diversas linguagens de programação. Contudo, grande parte desses *scripts* foram feitos utilizando a linguagem *Perl.*

Desse modo, diversos softwares foram produzidos utilizando o *CGI*: motores de buscas, sistemas de compras, reservas de ticket, entre outros. O poder que o *CGI* deu à *internet* foi gigantesco, impulsionando os sistemas *web* e mudando a forma como o usuário interagia com eles [1].

Apesar de tudo, o *CGI* possuía alguns problemas, sendo sua desempenho, talvez, o maior deles [1]. Naquela época não existia nenhum mecanismo capaz de manter uma mesma instância de um programa sendo executado no servidor por mais de uma requisição [1], ou seja, para cada nova requisição *HTTP*, o servidor tinha que executar uma nova instância do programa a ser utilizado. Isso causa grandes problemas de desempenho, já que essa tarefa é muito custosa para os sistemas operacionais [2].

2.1.2 FastCGI

Em meados dos anos 1990, surgiu o *FastCGI*, uma extensão do *CGI* que permite que diferentes requisições *HTTP* sejam atendidas por um único processo no sistema operacional, aumentando drasticamente a vida útil de cada processo e evitando reinicializar todos os processos e dependências a cada nova requisição *HTTP* [34]. Outra solução foi a criação de módulos que permitem que o servidor execute programas a partir do próprio processo no sistema operacional, eliminando também a mesma necessidade de criar novos processos a cada requisição [34]. Surgiu também o *Java Servlets*, essa tecnologia permite que várias requisições sejam atendidas por apenas um processo da *JVM* [2]. A cada nova requisição, um *thread Java* é criado e é responsável por processar essa requisição [2]. Essa técnica permite que os sistemas tenham uma escalabilidade bem melhor que os sistemas *CGI* clássicos, além de outros benefícios providos da linguagem *Java* [2].

Outro problema do modelo de sistemas *CGI* era a necessidade dos programas de produzir arquivos, geralmente *HTML* [1]. Dessa forma, o código dos programas continha grandes quantidades de fragmentos de texto e dados, tornando difícil a integração de designers nos projetos, já que muitos não possuíam o conhecimento técnico de programação necessário para trabalhar diretamente no código fonte desses programas [1]; então que surgiu o *Templates Systems (figura 5),* esse modelo permite que códigos script sejam injetados diretamente nos arquivos estáticos. Muitas tecnologias emergentes, como Microsoft ASP, Java *server pages* e PHP empregaram essa técnica. A técnica (oferecida por muitos processadores de texto) é similar à tecnologia já existente na época de criação de cartas dinâmicas, nas quais podemos ter um modelo de carta genérico e modificar apenas pequenos pontos específicos [1]. Esse modelo permite que *designers* e programadores trabalhem mais

harmonicamente nos sistemas, além de prover uma melhor separação entre as camadas de lógica e apresentação [1].

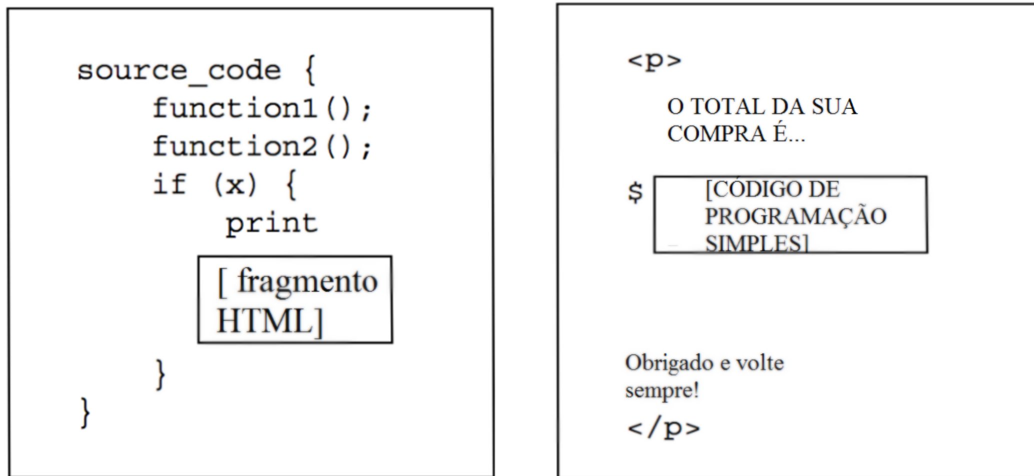

Figura 5: Comparação de uso entre programas CGI e *template systems*
Fonte: Web development with JSP, 2011, p. 6.

2.1.3 AJAX

AJAX (do inglês, *Asynchronous JavaScript and XML)* é um conjunto de técnicas de desenvolvimento web que permite que requisições assíncronas sejam enviadas aos servidores e respondidas, sem a necessidade de recarregar toda a página (Ullman 2007).

Até então, o funcionamento da web se dava da seguinte forma: os clientes enviavam dados dos seus browsers para os servidores, estes por sua vez, captavam esses dados, processavam e geravam uma página web que, então, era enviada de volta para os clientes [3]. A cada novo clique esse ciclo tinha que se repetir por completo, mesmo que somente uma pequena mudança na página fosse requisitada. Isso causa lentidão e frustração aos usuários, ao mesmo tempo, limita muito os sistemas *web*, quando comparados com sistemas desktop tradicionais [3].

Com o surgimento do *AJAX*, a interação do usuário ficou mais parecida com a dos sistemas *desktop* e as páginas não precisavam ser completamente recarregadas a cada novo clique [3], como representado na figura 6. Assim, muitos sistemas migraram dos sistemas tradicionais para os sistemas web [3] .

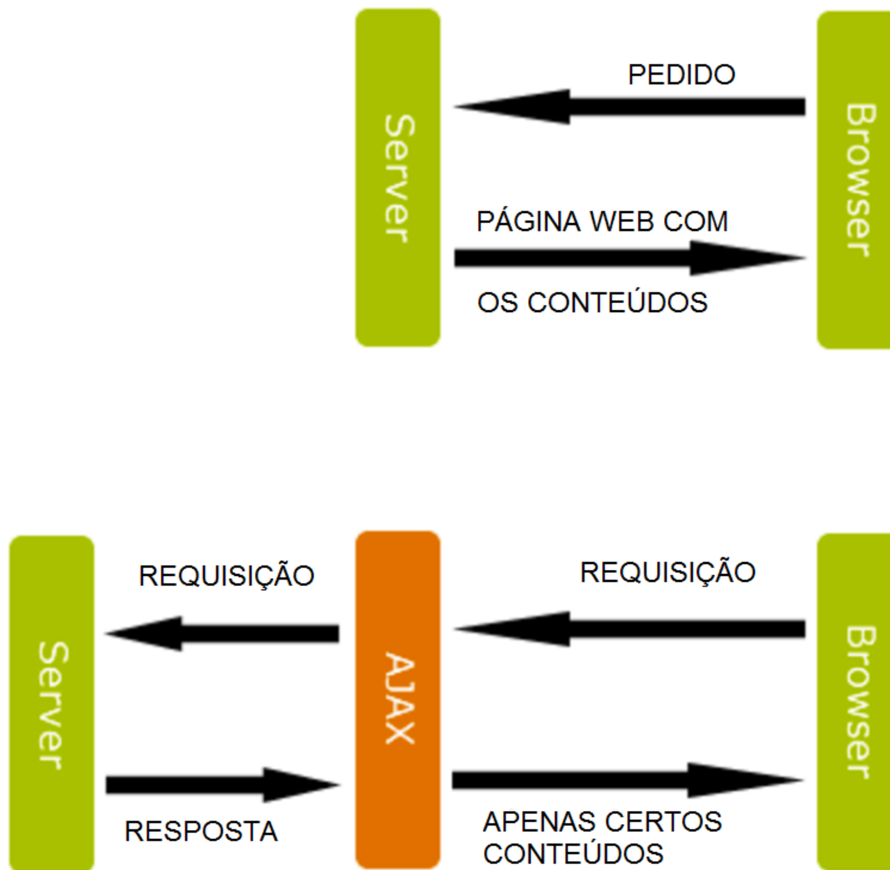

Figura 6: Modelo tradicional vs AJAX

Fonte:http://en.blog.bettr.info/index.php/2008/11/07/ajax-or-the-new-interaction-of-web-applications/. Acesso em: 9 ago. 2017

2.2. Arquitetura de sistemas

Desde os anos 1980, a arquitetura de sistemas tornou-se uma das mais importantes áreas de estudo no contexto de sistemas de larga escala [16]. A arquitetura de software engloba a análise de ferramentas e técnicas para se construir - da melhor forma possível - sistemas complexos de software. A arquitetura de softwares passou por uma transição, em que, antes, considerada uma área voltada à pesquisa, agora compõe essencialmente o processo de design e construção de softwares. (SHAW e CLEMENTS, 2006).

2.2.1 Arquitetura Cliente-Servidor

Em uma arquitetura cliente-servidor, a aplicação é modelada como um conjunto de serviços que são fornecidos por servidores [17]. Os clientes podem então acessar esses serviços e apresentar os resultados para os usuários finais (ORFALI e HARKEY, 1998). A arquitetura foi uma das primeiras arquiteturas de sucesso, e vem sendo utilizada até hoje. Ela passou por diversas transformações e evoluções, também servindo de base para diversas outras arquiteturas modernas [17].

2.2.1.1 Cliente-Servidor de duas camadas

Segundo Sommerville (2011), a arquitetura cliente-servidor de duas camadas é a forma mais simples da arquitetura cliente-servidor. O sistema é composto por um único servidor lógico e, um número indefinido de clientes que o utilizam. Esse modelo possui duas formas: o cliente magro e o cliente gordo.

No modelo cliente magro, somente a camada de apresentação é implementada no cliente e todo o resto do sistema é feito no servidor. Essas características oferecem algumas vantagens, sendo uma das mais evidentes a simplicidade do modelo e a facilidade de distribuição do software, já que toda a lógica do sistema é centrada no servidor. Por outro lado, o modelo pode apresentar altos custos de hardware, levando em conta que o sistema pode sofrer sobrecargas devido ao alto custo de processamento pela parte do servidor.

O modelo cliente "gordo" faz uso da capacidade de processamento dos computadores dos clientes, fazendo com que o servidor se torne essencialmente um serviço de gerenciamento de transações de banco de dados [4]. Essa característica faz com que a lógica nos servidores seja mais simples e tenha menos interações diretas com os clientes. Portanto,

esse modelo gera menos sobrecarga nos servidores. No entanto, o modelo dificulta o processo de alteração do sistema, uma vez que a cada nova alteração, o código dos clientes deve ser atualizado e redistribuído.

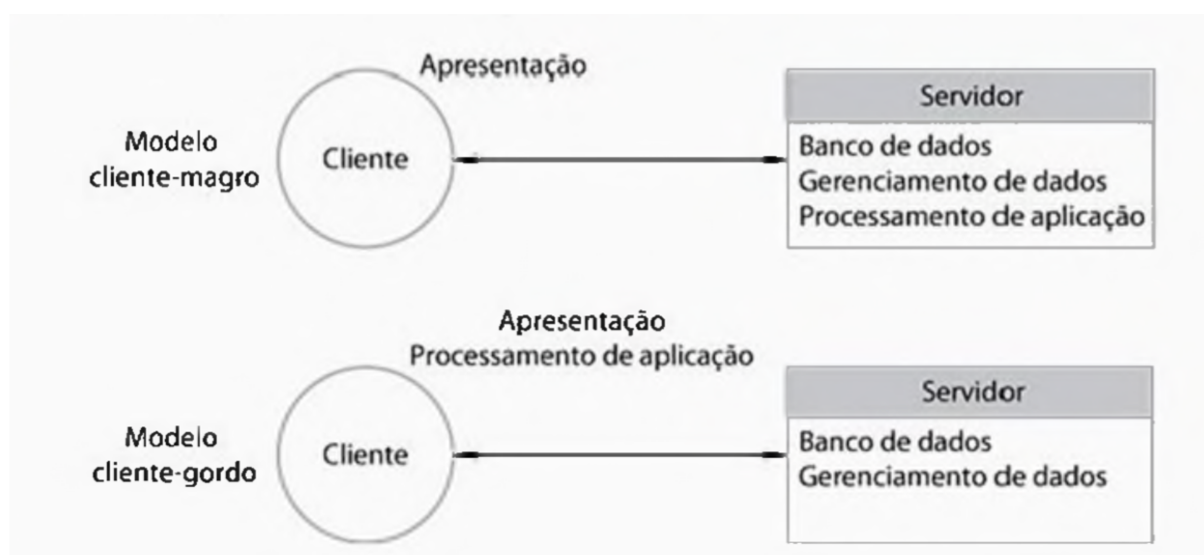

Figura 7: Arquitetura Cliente-Servidor de duas camadas
Fonte: Engenharia de software, 2011, p. 342**.**

2.2.1.2 Cliente-Servidor múltiplas camadas

O modelo cliente-servidor de múltiplas camadas resolve alguns problemas encontrados nos modelos mais simples [4]. Um deles sendo o fato de que as camadas lógicas do sistema - apresentação, processamento e banco de dados são mapeadas somente para duas camadas: o cliente e o servidor. Originando assim, problemas de escalabilidade, desempenho e gerenciamento do sistema [4].

Segundo Sommerville (2011), as arquiteturas multicamadas são inerentemente mais escaláveis, quando comparados aos de duas camadas. Assim, tornando o processo de aplicação do sistema mais fácil de ser atualizado, uma vez que, o processo é centralmente localizado na aplicação, como representado na figura 8. Tudo isso acarreta em respostas mais rápidas para as solicitações dos clientes e uma maior tolerância a problemas de escalabilidade e disponibilidade dos sistemas.

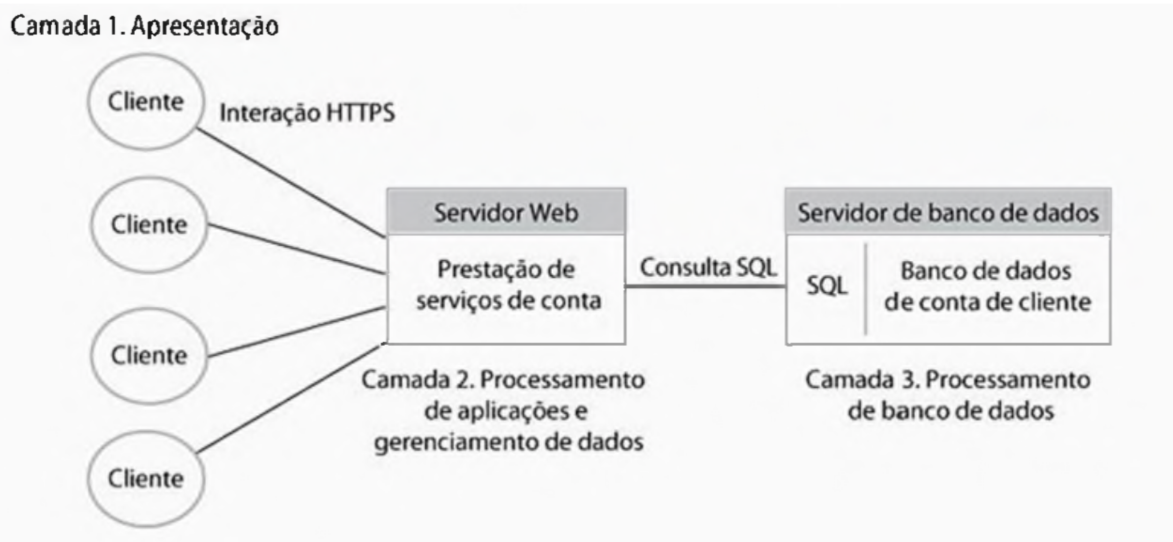

Figura 8: Exemplo de Arquitetura Cliente-Servidor de Múltiplas-camadas
Fonte: Engenharia de software, 2011, p. 344.

2.2.2 Arquitetura orientada a serviços (*SOA*)

A arquitetura orientada a serviços *SOA* (do inglês, *service oriented architecture*) é uma abordagem de design em que múltiplos serviços colaboram para fornecer algum conjunto final de recursos [5]. Serviços são processos completamente independentes e separados entre si. A comunicação entre esses serviços ocorre através de chamadas em uma rede em vez de chamadas de métodos dentro de um limite de processo [5].

De acordo com Newman (2015), essa técnica surgiu da necessidade de acabar com as dificuldades enfrentadas por arquiteturas tradicionais. Aplicações tradicionais tendem a ser grandes e monolíticas, gerando alto acoplamento e baixo reuso dos componentes do sistema. Com essa arquitetura, os sistemas podem ser mais rapidamente e independentemente produzidos, já que os componentes do sistema podem ser acoplados e desacoplados com mais facilidade. Um bom exemplo de uso dessa arquitetura seria o seguinte caso:

> *...Uma companhia aérea que pretende fornecer um pacote de férias completo para os viajantes. Assim como reservar seus voos, os viajantes podem também reservar hotéis no local de sua preferência, organizar o aluguel de carros ou reservar um táxi no aeroporto, procurar um guia de viagens e fazer reservas para visitar as atrações locais. Para criar essa aplicação, a companhia aérea compõe seu próprio serviço de reserva com os serviços oferecidos por uma agência de reservas de hotéis, aluguel de carro e companhias de táxi e serviços de reserva oferecidos pelos proprietários de atrações locais. O resultado final é um serviço único que integra os serviços de diferentes provedores" (SOMMERVILLE, 2011, p. 368).*

Figura 9: Exemplo do desacoplamento gerado pela arquitetura SOA
Fonte: Engenharia de software, 2011, p. 368.

2.2.2.1 Princípio fundamental da arquitetura SOA

Segundo Erl (2007), assim como o padrão de Programação Orientado a Objetos, a arquitetura voltada a serviços se tornou um padrão muito distinto, com princípios largamente aceitos que regem como a arquitetura deve ser implementada. Um sistema que segue a arquitetura SOA deve ter:

- **Baixo Acoplamento** - Os serviços devem manter uma relação mínima de dependência entre si.
- **Contrato de serviço** - Os serviços devem aderir a um acordo comum de comunicação entre os serviços.
- **Autonomia** - Os serviços devem ser capazes de controlar a lógica que encapsulam.
- **Reusabilidade** - A logica do sistema deve ser dividida em múltiplos serviços, para promover a reusabilidade.
- **Agregatividade** - Serviços devem ser capazes de se juntar e formar serviços compostos.
- **Sem Estado** - Serviços não devem informações estados de interações anteriores
- **Descoberta** - Os serviços devem ser projetados para serem encontrados e avaliados através de mecanismos de descoberta disponíveis.

2.2.2 Protocolos de comunicação

A comunicação entre os serviços que implementam a arquitetura *SOA*, deve ocorrer por meio da rede, ao contrário da arquitetura clássica, onde a comunicação entre os procedimentos ocorre dentro do mesmo processo (NEWMAN, 2015). Isso se dá por diversos problemas de integração enfrentados nas arquiteturas tradicionais. A integração entre os sistemas era pobre e difícil de ser implementada [5]. Cada sistema mantinha sua própria API, fazendo-se necessário entender cada uma e escrever um código de integração bastante único entre elas. Uma solução para esse problema foi criar uma camada de serviço que possuísse

um padrão único de comunicação [5]. Os mais conhecidos e utilizados são os padrões *SOAP* (do inglês, *Simple Object Access Protocol*) e *REST* (do inglês, *Representational state transfer*) [8].

Segundo Holdener (2008), SOAP é um protocolo baseado em XML utilizado para transmitir informações por meio de uma rede. SOAP é muito flexível, não dependendo de linguagens de programação específicas para funcionar.

O protocolo SOAP é composto de alguns elementos:

1. Envelope - Define onde a mensagem começa e termina
2. Corpo - Contém informações sobre a chamada ou resposta da requisição
3. Cabeçalho - Contém metadados da aplicação
4. Falha - Dados sobre possíveis tratamento de erros

O protocolo *SOAP* tem uma estrutura bastante formal, trazendo mais segurança e padronizando a forma de como os sistemas se comunicam [9]. É por isso que é comumente utilizado no mundo empresarial, onde grandes sistemas trocam mensagens a fim de realizar transações entre os seus sistemas [9]. O protocolo *SOAP* não define como a mensagem deve ser interpretada, em outras palavras, a semântica da requisição deve estar contida inteiramente dentro do corpo e do cabeçalho da mensagem, deixando os serviços que a utilizam totalmente responsáveis pela interpretação da mensagem [9].

Segundo Webber (2010), o formato *WSDL* (do inglês, *Web Service Definition Language*) é comumente utilizado para descrever interfaces de serviços na internet. Ele foi derivado do protocolo SOAP e é comumente utilizado junto com a linguagem *UDDI* (do inglês, *Universal Description, Discovery and Integration*), que por sua vez, é um tipo de registro baseado em *XML*. O *UDDI* permite que os sistemas listem seus serviços e os tornem visíveis e utilizáveis por outros sistemas.

REST é um método simples de transporte de dados, essencialmente voltado para a Web, embora não seja restrito a ela [3]. O *REST* surgiu como uma alternativa mais simples aos protocolos já existentes, tal como o *SOAP*. Diferentemente, o *REST* não possui camadas extras de mensagens ou padrões bem definidos de como as mensagens devem ser transportadas, simplificando muito sua estrutura [3]. Os conceitos chave para alcançar um sistema baseado na arquitetura REST são:

- Separação de estado e funcionalidade.
- Recursos devem fornecer métodos consistentes de transferência de estados.
- Recursos são acessíveis através de verbos HTTP.
- Protocolo não deve guardar estado de operações anteriores.

Diferentemente do SOAP, o protocolo REST deve, idealmente, utilizar os verbos HTTP - POST, GET, PUT e DELETE. Cada um desses métodos, implicitamente injeta uma semântica diferente em cada chamada [7]. A tabela abaixo define os principais verbos HTTP e sua função.

Verbo	Função
GET	Ler e recuperar dados
POST	Criar novos recursos
PUT	Atualiza recursos existentes
DELETE	Remove recursos

Fonte: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>, acesso em 02 de ago de 2017

Serviços baseados em *REST* requerem pouca infraestrutura além da já existente no protocolo HTTP [18]. Eles são geralmente simples e efetivos, já que o protocolo *HTTP* fornece uma grande variedade de interfaces que podem ser utilizadas por eles [18].

De acordo com Ricardo (2015), escolher entre *REST* e *SOAP* vai depender dos requisitos das organizações envolvidas nos processos. Se as transações requerem um nível mais elevado de segurança e integridade, o *SOAP* provavelmente se encaixaria melhor nos requisitos, pelo fato de possuir mais protocolos alinhados, gerando mais segurança, uma vez que existe uma preocupação maior com a segurança nesses protocolos. Por outro lado, se as transações devem ser mais leves e simples, o protocolo *REST* seria mais adequado.

2.2.2.3 Níveis de maturidade do *REST*

Segundo WEBBER (2011), os sistemas que utilizam *REST* como protocolo de comunicação podem ser classificados em quatro níveis de maturidade. Quanto maior o nível de maturidade, mais adaptados as características dos sistemas *REST* eles estão, são eles:

- **Nível 0:** É o nível mais básico, é fornecido aos consumidores uma *API* que pode ser chamada via *HTTP*, embora não utilize nenhum mecanismo específico da *WEB*. Essencialmente é feita uma chamada remota de procedimento via *HTTP*.
- **Nível 1 – Recursos**: Esse nível basicamente faz o mesmo do nível zero, com uma diferença: Ao invés de chamar uma função qualquer via *HTTP*, o sistema endereça os

recursos diretamente, ou seja, existe uma separação mais explicita do nicho de cada requisição.

- **Nível 2 – Verbos HTTP:** No nível 0 e 1 o verbo *POST* é utilizado para todas as chamadas. Nos sistemas nível 2, os verbos *HTTP* devem ser utilizados com a maior proximidade possível do que eles representam, por exemplo: deve se usar *GET* para requisições somente leitura, *DELETE* para remoção de dados, e etc. Dessa forma, é possível categorizar a natureza de cada requisição somente verificando qual verbo *HTTP* é utilizado.
- **Nível 3 – Controles de Hipermídia:** Nesse nível os sistemas devem explicitar para os seus consumidores o que o sistema é capaz de fazer depois de cada requisição. Por exemplo, dentro do corpo da resposta das requisições, o sistema deve fornecer caminhos que possam ser utilizados pelos clientes para efetuar outras tarefas ou até mesmo continuar e finalizar a tarefa corrente, permitindo assim que haja uma maior flexibilidade na comunicação entre os sistemas. Essa estratégia permite que novas funcionalidades sejam modificadas, adicionadas e removidas sem que haja um grande impacto no código existente. Esse nível também fornece uma forma fácil para que os consumidores dos serviços descubram o que os sistemas são capazes de fazer.

2.2.2.4 Popularidade do *REST*

A popularidade dos serviços baseados em *REST* aumentou muito desde sua criação [18]. Em contraste com o *SOAP* que vem sendo cada vez menos utilizado, restringindo-se basicamente ao mundo empresarial [36]. Isso se dá pela complexidade de se utilizar o *SOAP*, *pois, nesse caso,* os sistemas têm que seguir regras e padrões muito estritos, e muitas vezes, complicados. Já o *REST* é facilmente e rapidamente implementado, pelo fato de utilizar os protocolos nativos da *W*eb e não requerer uma estrutura rígida nas suas mensagens [36]. Outro fator que fez o *REST* se destacar é o crescente número de *API's* existentes na *Web.* Normalmente, as *API's* devem ser simples e fáceis de usar, resultando na predominância do *REST* em suas implementações.

2.2. 3. Arquitetura de Micro-serviços

Ao longo dos anos, a indústria do software vem tentando construir sistemas de informação cada vez melhores [5]. Aprendendo com sistemas já existentes, adotando novas tecnologias e observando como as empresas empregam essa tecnologia, tudo isso com propósito de servir os seus clientes melhor [5].

Segundo Newman (2015), os Micro-serviços originaram-se de um ambiente onde eram empregados conceitos como: arquiteturas voltadas ao domínio (do inglês, *Domain-*

driven design), entregas incrementais, virtualização, automação da infraestrutura, grande escalabilidade e pequenos e independentes times de desenvolvimento. Todas essas características, deram origem ao que conhecemos hoje como a arquitetura de Micro-serviços. Segundo a definição formal de Newman (2015), Micro-serviços são pequenas entidades independentes (serviços) que trabalham de uma maneira colaborativa dentro de um sistema maior.

A arquitetura de Micro-serviços compartilha muitas características da arquitetura *SOA*, de fato, elas caminham juntas.

> *"... Você deve pensar na arquitetura de Micro-serviços como uma especialização do SOA, da mesma forma que Scrum ou XP são especializações do método de desenvolvimento ágil de software" (NEWMAN, 2015, p. 9).*

As arquiteturas tradicionais tendem a se tornar muito grandes e monolíticas, fazendo-se surgir muitos problemas: dificuldade em localizar erros, duplicidade de código, entre outros.

Micro-serviços possuem barreiras bem definidas, tornando-se fácil identificar onde o código responsável por cada tarefa está localizado [5]. Além disso, serviços muito especializados e independentes tornam o sistema mais conciso e evita duplicações de código desnecessárias [5].

Idealmente cada serviço na arquitetura deve se comportar como uma entidade separada das restantes [5]. Os serviços devem ser capazes de rodar em plataformas diferentes, tais como: sistemas *AWS* da Amazon, *Windows Azure*, Google *App Engine* etc. Essa característica torna o sistema mais distribuído e resistente a falhas; caso um dos serviços apresente problemas, os demais continuaram a funcionar corretamente, cada um em sua plataforma, como representado na figura 10. A comunicação entre esses serviços se dá por meio da rede, utilizando um protocolo comum e suas *API's*, resultando em baixo acoplamento [5].

Sistemas independentes trazem muitos benefícios, muitos deles muito relacionados com os dos sistemas distribuídos e arquitetura orientada a serviços [5]. Além disso, a arquitetura de Micro-serviços é capaz de utilizar diferentes tecnologias em cada serviço do sistema; dependendo dos requisitos, podemos escolher diferentes linguagens de programação. Por exemplo, um sistema pode escolher utilizar Java, quando precisa tratar grandes quantidades de dados ou Ruby para um desenvolvimento mais ágil [5].

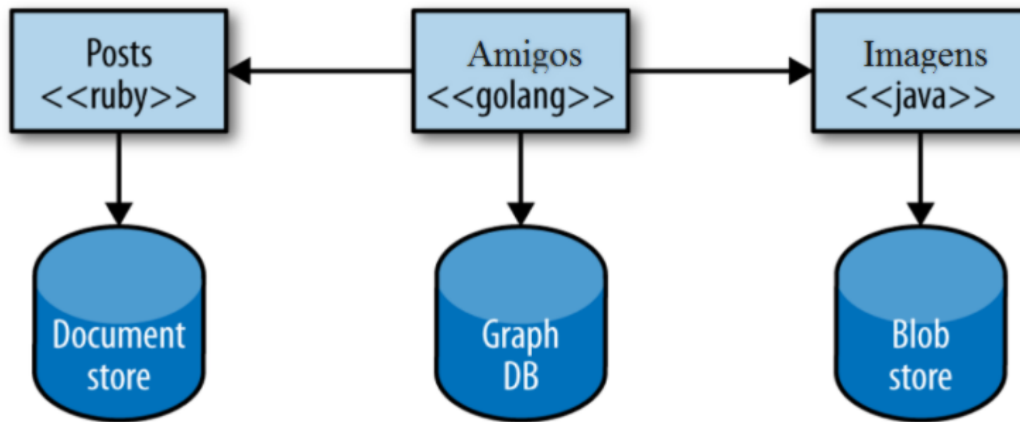

Figura 10: Diferentes tecnologias na arquitetura de Micro-serviços
Fonte: Building microservices, 2015, p. 4.

Outra grande característica dessa arquitetura é a capacidade de escalar os sistemas de uma forma mais individual e organizada [5]. É muito difícil aplicar escalabilidade específica em sistemas grandes e monolíticos - se somente uma determinada parte desse sistema precisa de mais desempenho, todo o sistema deve ser escalado. Não existe a opção de escalar partes individuais do sistema. Na arquitetura de Micro-serviços isso é relativamente simples de ser feito, já que cada serviço é executado em máquinas separadas [5]. Com isso, podemos escalar os sistemas críticos de forma individual e utilizar os serviços mais simples em máquinas menos poderosas e mais baratas. Dessa forma, utilizando o exemplo anterior, poderíamos ter várias instâncias do mesmo serviço lidando com o processamento intenso de imagens e menos instâncias para servir as partes menos críticas do sistema (Figura 11).

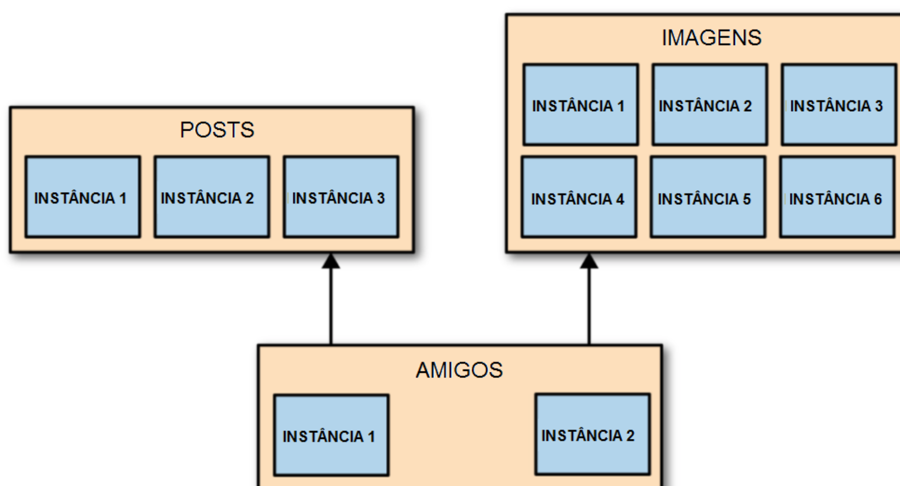

Figura 11: exemplo de escalabilidade específica no Micro-serviços
Fonte: Building microservices, 2015, p. 6.

Cada vez mais observamos uma popularização das arquiteturas distribuídas baseadas em serviços [5]. Virtualização, escalabilidade horizontal, infraestruturas resilientes e redundantes são de extrema importância para grandes serviços, com isso as tradicionais arquiteturas em camadas vêm perdendo espaço [5]. Grandes empresas como Google, Amazon e *Netflix* estão adotando cada vez mais os Micro-serviços [5].

3. Arquitetura MVC

O padrão MVC (do inglês, *Model view conroller*) foi inventado por Trygve Reenskaug, enquanto fazia uma visita a um grupo de pesquisa da então Xerox Palo Alto, centro de pesquisa [12]. O seu primeiro artigo sobre o tema foi publicado em 1978 e originalmente chamou o padrão de *Thing-Model-View* (ou coisa-modelo-visão), logo após o nome foi trocado para *Model-View-Controller* (ou modelo-visão-controlador) [12].

De acordo com Walther (2018), Reenskaug tentava resolver o problema de representar modelos complexos do mundo real, tais como a construção de uma ponte, estação de energia ou uma plataforma de extração de petróleo. Reenskaug procurava uma forma de interligar os diferentes modelos de representação desses problemas - modelo mental e computacional [12]. Tais modelos eram bastante diferentes. Os seres humanos têm uma representação mental dessas estruturas, já os computadores, possuem uma representação digital desses problemas. Para isso, ele definiu a estrutura básica de organização do MVC - modelo, visão e controlador [12].

O modelo MVC sofreu várias mudanças durante sua evolução, e foi se adequando a outros modelos e arquiteturas de sistemas. O modelo é utilizado desde as tradicionais aplicações monolíticas até os sistemas mais desacoplados, como os da arquitetura de *SOA* e Micro-serviços [17].

> *"...Como se verifica, as estruturas MVC também são adequadas para o desenvolvimento de end-points REST. A natureza orientada a recursos do REST mapeia bem o conceito de controladores e modelos" (SLATER, N. 2015)*

3.1 Proposta Original do MVC

O modelo MVC passou por diversas mudanças ao longo dos anos. Os sistemas de informação estavam apenas começando a utilizar as interfaces gráficas, aumentando a complexidade e a forma de se fazer sistemas. Então, se viu necessário uma arquitetura capaz de separar e organizar melhor as entidades do sistema [11]. Inicialmente a proposta do MVC era a seguinte:

- **Modelo -** Representa alguma informação do sistema.
- **Visão -** Uma representação visual do modelo. O mesmo modelo poderia ter várias diferentes representações, as visões devem se relacionar com os modelos pelo padrão *Observer*
- **Controlador -** Deve coletar entradas de usuários e modificar os modelos

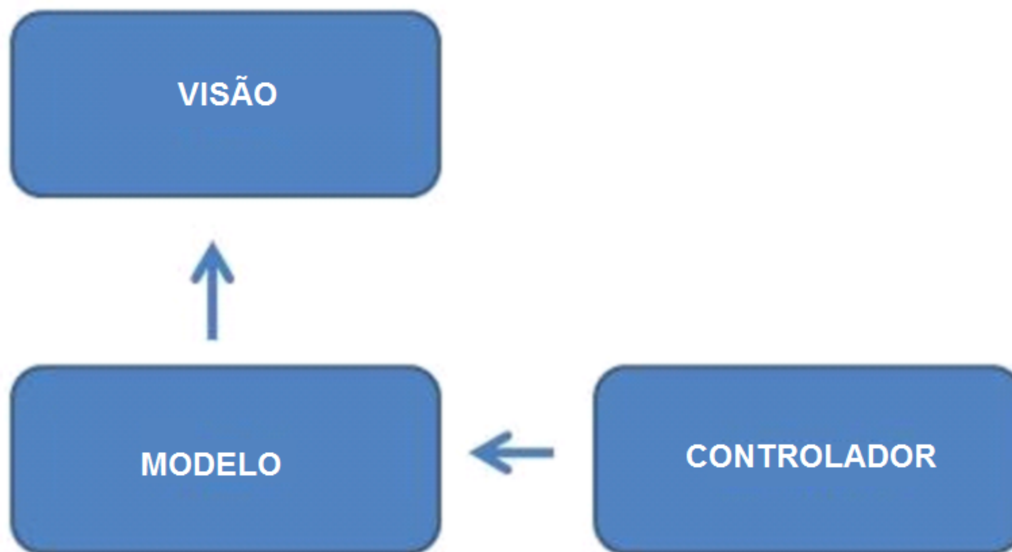

Figura 12: Modelo MVC original
Fonte: http://stephenwalther.com/archive/2008/08/24/the-evolution-of-mvc. Acesso em: 7 ago. 2017

A arquitetura prega que a camada de visão deve ser atualizada indiretamente pelo modelo. A cada mudança nos dados, um evento é gerado no modelo e as visões que estão ligadas a ele são atualizadas [11] .

Por sua vez, o controlador não deve se comunicar diretamente com as visões, deve notificar o modelo que por sua vez notifica a visão, fazendo com que ela seja atualizada. Em síntese a arquitetura deve seguir a ideia:

> *"Todo código que manipula interfaces de apresentação deve apenas manipular interfaces de apresentação, deixando toda a lógica do domínio e o acesso a dados do sistema em suas determinadas áreas do programa" (MARTIN, 2006).*

3.2 Modelo 2 do MVC

O conceito de MVC mudou radicalmente com a introdução da tecnologia *JSP* (do inglês, *Java Server Pages*). A especificação dessa tecnologia incluía dois métodos de como projetar as aplicações: o Modelo 1 (MVC original) e o Modelo 2 [12]. Percebe-se que, em sistemas voltados para a Web, é mais comum que tenhamos o Modelo tipo 2 [12].

- **Modelo -** Representa lógica de negócio e acesso a camada de dados (banco de dados relacional por exemplo).

- **Visão -** Camada de apresentação de dados
- **Controlador -** Meio pelo qual o usuário interage com a aplicação

No modelo 2, não existe mais relação direta entre a visão e o modelo. Toda a comunicação entre eles acontece por meio do controlador, representado na figura 13 [12].

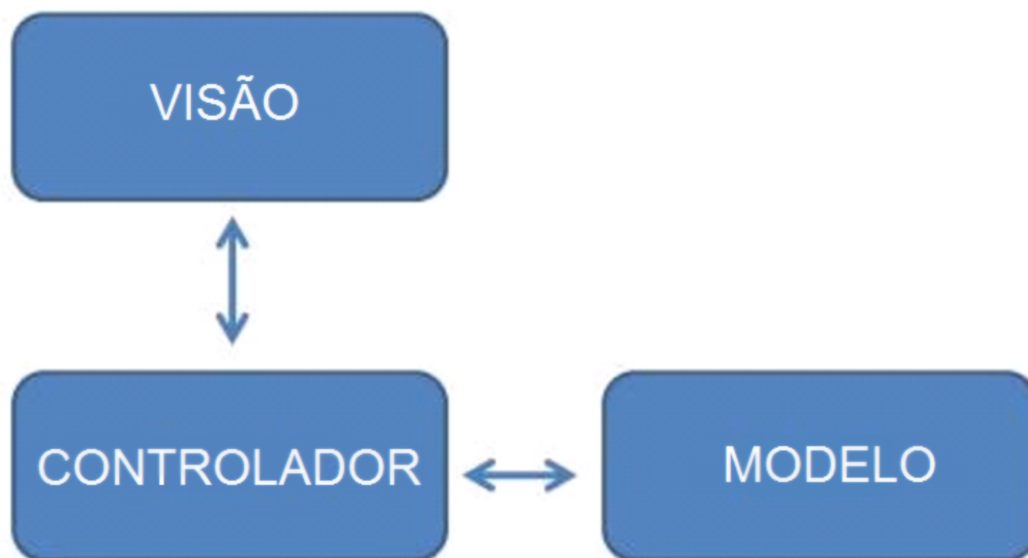

Figura 13: Modelo 2 do MVC
Fonte: http://stephenwalther.com/archive/2008/08/24/the-evolution-of-mvc. Acesso em: 7 ago. 2017

3.3 Arquitetura *MVP*

A arquitetura MVP (do inglês, *Model View Presenter*) é, ao mesmo tempo, uma evolução do MVC e também outra maneira de a interpretar [13]. Ela é composta pelas camadas: modelo, visão e apresentação (análogo ao controlador do MVC). A principal diferença entre as arquiteturas é o fato de que, na arquitetura MVP, a camada de visão e a de apresentação são interligadas de uma forma diferente do MVC [13]. No MVC, a camada de visão se comporta de uma forma mais independente, já no MVP, a visão deve delegar qualquer evento disparado nele para a sua camada de apresentação (figura 14). Outra diferença é que a camada de visão é conectada direta ou indiretamente ao seu modelo (dependendo da versão do MVC), já no MVP, a visão e o modelo são completamente independentes [13].

- **Modelo -** Representa lógica de negócio e acesso a camada de dados (Mesmo conceito na arquitetura MVC).

- **Visão -** Camada de apresentação de dados. Ele deve diretamente referenciar a sua respectiva camada de apresentação.
- **Apresentação -** Deve guiar a lógica da camada de visão. Essa camada deve conhecer a visão e o modelo. Ela atualiza a camada de visão, baseada em mudanças ocorridas no modelo, deve também atualizar o modelo, baseado em eventos disparados pela visão. Essa camada deve encapsular toda a lógica de apresentação e define os valores e métodos que a camada de visão deve utilizar.

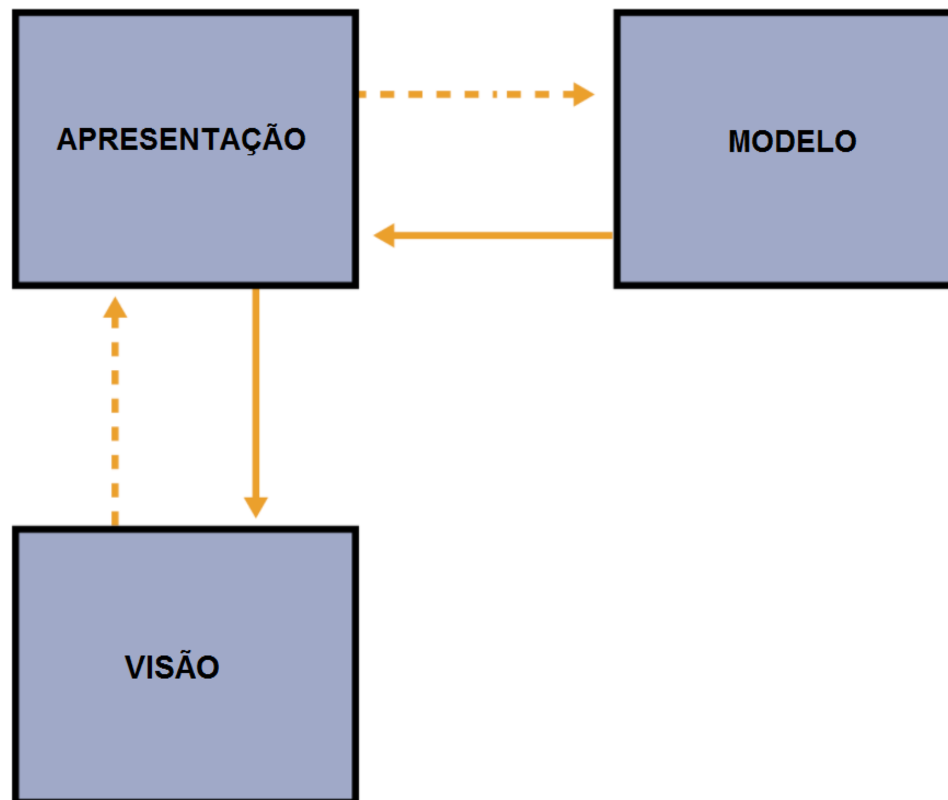

Figura 14: Arquitetura MVP

Fonte: Building enterprise applications with Windows Presentation Foundation and the model view viewmodel pattern, 2011, p.35

3.4. Arquitetura MVVM

O padrão MVVM (do inglês, *Model-View View Model*) é uma arquitetura baseada nas arquiteturas MVC e MVP, oficialmente lançada em 2005 por John Grossman [22]. O *MVVM* propõe uma separação forte entre a camada de visão e o modelo através de um mecanismo de ligação de dados dinâmico (do inglês, *data-binding*) entre a camada de visão e a visão-modelo [22]. Esse mecanismo permite que eventos de mudanças no sistema sejam refletidos com muita facilidade na camada de visão [22]. Outro grande benefício da arquitetura é que a

separação entre essas camadas permite que os componentes Modelo e Modelo-Visão sejam reutilizados completamente em projetos distintos, sendo necessário somente uma camada de visão diferente para que o sistema seja portável para diferentes aplicações [22]. Essa arquitetura foi criada principalmente para facilitar a atualização das camadas do sistema a cada evento disparado, facilitando o desenvolvimento de aplicações e evitando códigos específicos e repetitivos para lidar com o tratamento de eventos dentro do sistema [14].

- **Modelo -** Representa lógica de negócio e acesso a camada de dados (Mesmo conceito na arquitetura MVC).
- **Visão -** Contém os elementos de apresentação de dados. Deve se comunicar com a camada Visão-modelo utilizando *data-binding,* comandos e notificações.
- **Visão-Modelo -** Contém a lógica de apresentação, e provê dados para a camada de visão, se comunica diretamente com a visão por meio de *data-binding*, comandos e notificações. Deve ser capaz de atualizar dados na camada modelo

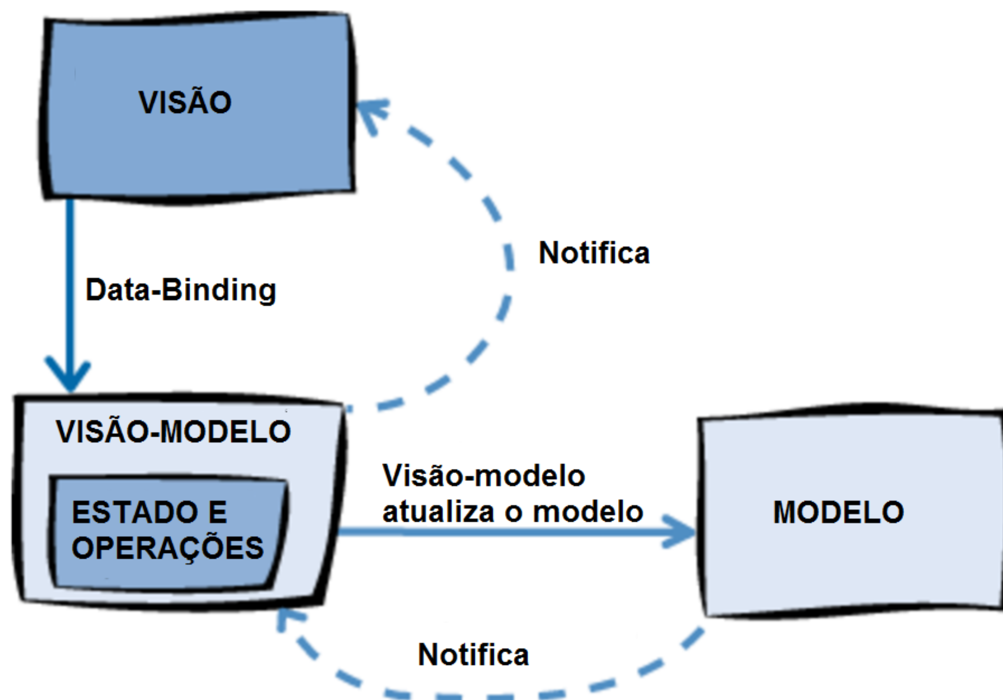

Figura 15: Arquitetura MVVM
Fonte: https://msdn.microsoft.com/en-us/library/hh848246.aspx. Acesso em: 7 ago. 2017

4. Frameworks Web

A evolução dos sistemas *Web* ao longo dos anos trouxe com eles um grande aumento de complexidade das suas arquiteturas [15], abrindo espaço para uma infinidade de problemas decorrentes dessa evolução, afirma Ford (2004). Foi então que o uso de frameworks se tornou um hábito comum no processo de desenvolvimento desses sistemas.

Segundo Ford (2004), um framework de desenvolvimento é um conjunto de classes, bibliotecas e outros tipos de elementos, que visa facilitar o desenvolvimento de aplicações, fornecendo elementos completa ou parcialmente prontos. Os frameworks, geralmente, disponibilizam elementos bastante genéricos, tais como: ferramentas de *login*, utilitários para acesso a banco de dados, mapeamento de *URL*, gerenciamento de sessões, entre outros [15]. Além disso, frameworks fornecem um ciclo de atividade padrão para que o programador siga durante o desenvolvimento do produto [15]. No caso dos frameworks MVC, os programadores devem respeitar os componentes da arquitetura – Modelo, Visão e Controlador e seguir a arquitetura corretamente, tendo como benefício um desenvolvimento mais rápido e limpo, favorecendo muito a manutenabilidade do sistema [15].

4.1 CodeIgniter

O framework Codeignter é um framework de código aberto voltado para o desenvolvimento de aplicações *web* na linguagem *PHP*. Criado em 2006 por Rick Ellis, segue o padrão MVC (embora não force o programador a segui-lo estritamente) e fornece diversas bibliotecas e ferramentas para auxiliar no desenvolvimento das aplicações [20]. Como citado anteriormente, as classes tipo modelo do framework são responsáveis por gerenciar acesso ao banco de dados da aplicação e efetuar a lógica de negócio do sistema [20]. A visão será responsável por apresentar dados aos usuários, no CodeIgniter elas são representadas normalmente por páginas HTML, mas podem ser também uma página do tipo RSS [20].

Por fim, os controladores servem como intermediários entre a visão e o modelo, eles serão responsáveis por receber solicitações, encaminhar requisições e mandar respostas de volta para os clientes [20]. Além disso, o framework disponibiliza uma grande variedade de bibliotecas e classes auxiliares prontas para uso, tais como: *caching*, medição de desempenho, gerenciamento de arquivos, paginação, sessão, cookies, segurança, gerenciamento de e-mails etc.

Além disso, se o framework não disponibilizar o recurso nativamente, o programador pode adicionar bibliotecas extra ao seu framework [21].

O ciclo básico de uma requisição HTTP utilizando o framework é representado na figura 16:

1. A requisição do cliente atinge o arquivo base do framework (index.php) e é então redirecionado para o componente roteador.
2. O roteador vai então, decidir se irá utilizar uma página do cache ou se irá processar a requisição pelo fluxo comum.
3. Caso a requisição tenha que ser processada, o módulo de segurança realiza algumas checagens, e então redireciona a requisição para o controlador correto.
4. A partir daí o controlador pode utilizar diversos elementos para processar a requisição.
5. Geralmente o resultado da requisição vai ser gerado em forma de uma visão que vai ser retornada para o cliente.

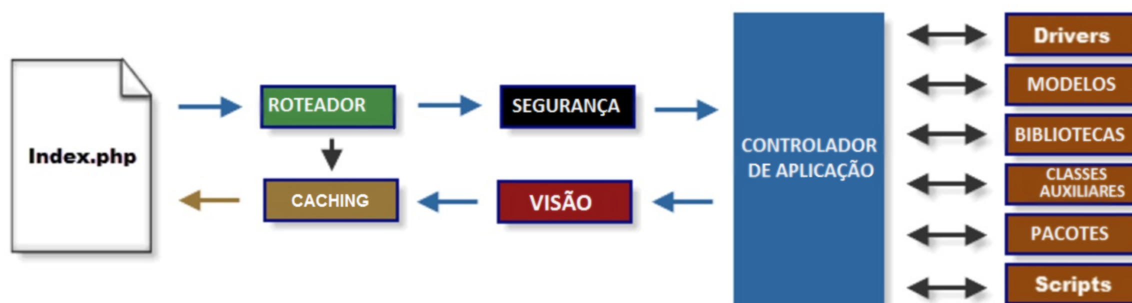

Figura 16: Ciclo de vida do CodeIgniter
Fonte: https://www.codeigniter.com/user_guide/overview/appflow.html. Acesso em 20 Ago, 2017

4.2 KnockoutJS

Existem também frameworks para desenvolvimento *Web* voltados para o lado do cliente, onde é dada mais ênfase nos eventos e na experiência de usuário. KnockoutJS é um grande exemplo de um desses frameworks [22].

Segundo MUNRO (2014), KnockoutJS é um framework *open-source,* baseado em *Javascript*, criado para que os programadores conseguissem produzir rapidamente aplicações *Web* ricas e dinâmicas para o *front-end*. Utilizando o padrão *MVVM*, KnockoutJS torna muito simples a implementação de complexas interfaces de usuários, tornando as aplicações extremamente amigáveis e eficientes.

A mais importante característica desse framework é a capacidade de realizar *data-bidings (*ligação de dados*)* entre a camada visão e a modelo-visão [22]. Dessa forma, podemos, de uma forma transparente ao programador, refletir mudanças da camada de visão diretamente nos modelos e vice-versa.

KnockoutJS é um framework bastante simplificado, focado basicamente em ligação de dados entre as camadas. Com isso, ele se adequa muito bem a outros frameworks [22], por exemplo: poderíamos utilizar o KnockoutJS juntamente com um framework focado no lado do servidor, dessa forma, teríamos todo o poder de processamento de dados do servidor, aliado a uma camada simples e poderosa no lado do cliente, essa comunicação é representada na figura 17.

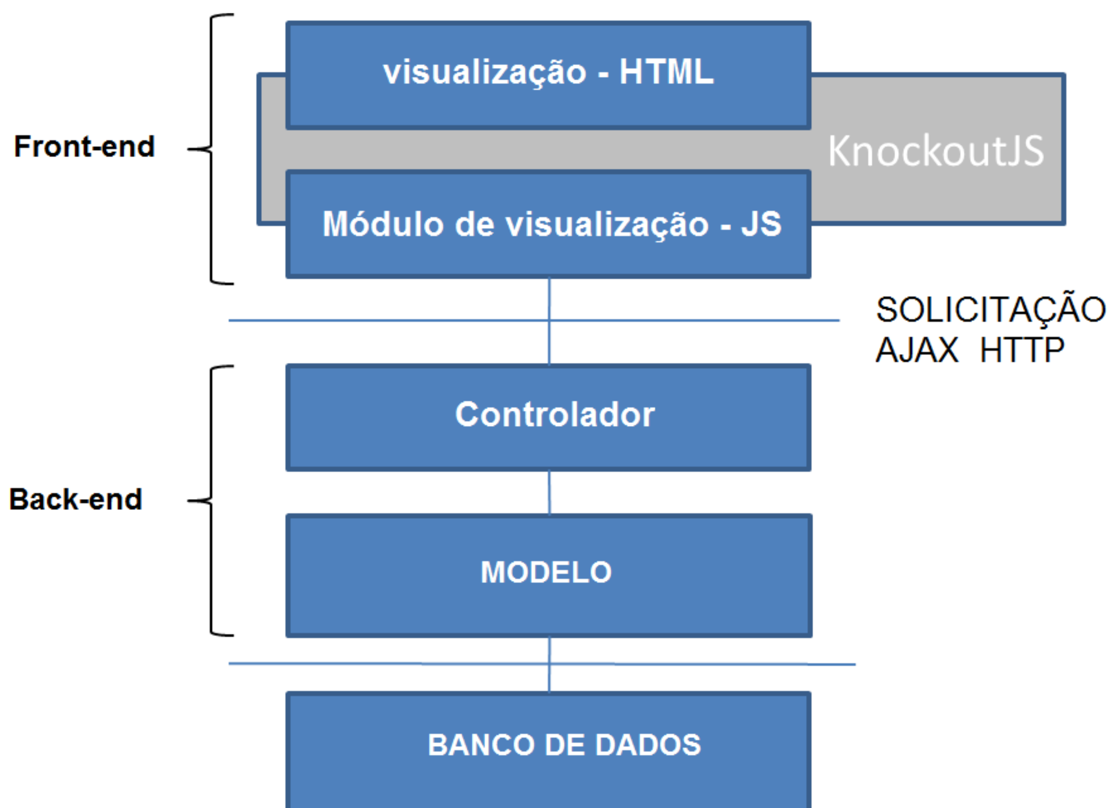

Figura 17: Interação de frameworks *front-end e back-end*

Fonte: https://www.codeproject.com/Articles/568115/Sample-application-RavenDB-KnockoutJS-Bootstrap-We. Acesso em: 11 set 2017

5. *Single Page Web-Application*

Cada vez mais as aplicações tradicionais de sistemas *Web* vêm sendo substituídas por arquiteturas conhecidas como *SPA* (do inglês, *single page application*). Esse tipo de sistema é caracterizado pelo fato de não necessitar recarregar todo o sistema a cada requisição feita pelo usuário [47]. Em outras palavras, as aplicações são carregadas em um momento inicial, e a medida que o sistema é utilizado a interface do sistema é atualizada dinamicamente; solicitando somente o necessário dos servidores. As primeiras tecnologias *Web* a fornecer essa arquitetura foram as de *Flash* e Java *applets*. Elas dominavam o mercado de sistemas Web que utilizavam a arquitetura *SPA* até que os frameworks Javascript cresceram e se popularizaram, resultando na quase extinção da utilização do Flash e Java *applets* 47].

Existem inúmeros benefícios de se utilizar essa arquitetura, entre eles, a usabilidade e fluidez de uma aplicação desktop combinado com a portabilidade e acessibilidade de um sistema *Web* [47].

- **Renderização de partes específicas:** a aplicação tem a capacidade de redesenhar somente o necessário da interface quando o usuário efetua uma ação. Por outro lado, um sistema *Web* tradicional, redesenha toda a página, muitas vezes para atualizar uma pequena porção da interface, resultando em uma pausa na usabilidade para que a página seja completamente refeita pelo servidor e transferida para o cliente. Dependendo da qualidade da conexão, do tamanho da página e do trafego atual da rede, essa pausa pode ser bastante longa, resultando em uma péssima usabilidade do sistema.
- **Melhor tempo de resposta:** a aplicação responde mais rapidamente a eventos disparados pelo usuário, pois grande parte do processamento é repassado para o lado do cliente, evitando o envio requisições para o servidor, processamento e resposta.
- **Utilização de menos recursos do servidor:** com o uso de *SPA* o servidor reduz a quantidade de dados processados e enviados pela rede, pelo fato de que o sistema não precisa recarregar continuamente atributos estáticos do site, tais como: menus, cabeçalho, rodapé, arquivos de estilo, arquivos *Javacript*, entre outros.

6. PHP

PHP (do inglês, *Hypertext Preprocessor*, originalmente *Personal Home Page*) é uma linguagem de programação interpretada originalmente desenvolvida em 1994 para desenvolvimento de aplicações web ao lado do servidor [23]. Ela também pode ser utilizada como linguagem de linha de comando e também para produzir interfaces gráficas, embora o maior foco da linguagem seja para desenvolvimento *WEB* [23].

Segundo Tatroe (2014), O *PHP* passou por drásticas mudanças. Inicialmente era simplesmente um conjunto de funções auxiliares escritas em C que auxiliavam os servidores a trabalhar no início da WEB através de interfaces de CGI. Logo depois, o PHP passou a ser capaz de interagir com banco de dados e gerar páginas mais dinâmicas para a web. A partir daí o PHP ficou extremamente popular e várias outras versões foram lançadas [24]. O PHP hoje suporta técnicas modernas de desenvolvimento, tais como: orientação objeto, *namespaces*, programação funcional, *opcode cache*, entre outras [24]. Hoje o PHP é visto como a linguagem mais popular da web, presente em 82.8% dos servidores [25] e é utilizado desde as mais simples páginas web até os mais elaborados serviços, tais como: Facebook, Baidu.com, Wikipedia.org entre outros [25].

O desenvolvimento de aplicações complexas em PHP geralmente é feito utilizando-se algum framework [35]. Atualmente no mercado encontramos um grande leque de opções, cada um especializado em um tipo de desenvolvimento – Aplicações ERP, Aplicações voltadas a *endpoints,* blogs, entre outras. Os mais conhecidos frameworks PHP são: *Laravel*, *Synfony*, *Codeigniter*, *CakePHP*, entre outros [35].

7. Banco de Dados

Sistemas de banco de dados são essenciais em quase todos os sistemas de informação [26]. Os bancos de dados armazenam, organizam e tratam os dados que norteiam as aplicações [27]. O uso desses sistemas aumenta a eficiência com que os dados são gerenciados, diminuem custos e centralizam o conhecimento dos sistemas, permitindo que diversas técnicas de análise de dados sejam aplicadas, a fim de gerar um conhecimento potencialmente valioso sobre os negócios [26].

Segundo SILBERSCHATZ (2011), esses sistemas passaram por drásticas mudanças ao longo do tempo:

- **1950's e 1960's:** fitas magnéticas eram utilizadas para o armazenamento de dados, simples tarefas como folha ou dados estatísticos eram realizadas utilizando esses cartões.
- **1970's:** o uso de discos rígidos para o armazenamento se tornou mais abrangente, agora os dados eram acessados muito mais rapidamente. Agora, sistemas de banco de dados hierárquicos puderam ser criados
- **1980's:** durante os anos 1980, os sistemas de banco de dados relacional passou por grandes mudanças, resultando em novas tecnologias e sistemas muito rápidos.
- **1990's:** bancos de dados Orientados a objeto ganharam muita força, e também tiveram que acompanhar o surgimento de sistemas web, dessa forma, os sistemas de banco de dados ganharam alto desempenho em e alta taxa de disponibilidade.
- **2000's:** a grande ascensão dos sistemas de banco de dados *open-source*, particularmente, PostGreSql e MySql. Além disso, um novo paradigma de banco de dados se popularizou: os banco de dados *NoSQL* . Esses sistemas são ideais para sistemas que lidam com uma grande quantidade de dados. Fazendo-se prefeito para grandes sistemas, tais como: Amazon, Facebook, Google, entre outros. Também, novas técnicas de *data-mining* foram desenvolvidas para que se pudesse extrair conhecimentos valiosos dessa grande massa de dados. Também surge o mais recente paradigma – *NewSQL,* que tenta combinar algumas características tradicionais do paradigma relacional e algumas vantagens dos sistemas *NoSQL.*

7.1 Bancos Relacionais

De acordo com SILBERSCHATZ (2011), o modelo de banco de dados relacional é baseado em tabelas, que representam tanto os dados quanto as relações entre esses dados. A simplicidade desse modelo o tornou tão popular, por isso, a maioria dos sistemas utiliza o modelo relacional [27]. Linguagens específicas foram desenvolvida para que os usuários interagissem com esse banco de dados, sendo a SQL a mais utilizada *(*do inglês, *Structured Query Language).*

A linguagem SQL é dividida em várias categorias:

- **DDL (do inglês, *data definition language*):** essa categoria é responsável para manipular a estrutura das tabelas, tais como : campos, tipo dos campos, nomenclatura, chaves primarias, chaves estrangeiras, entre outras.
- **DML (do ingês, *data manipulation language*):** essa categoria é responsável por manipular os dados contidos nas tabelas, os comandos permitem : inserir, remover e modificar linhas nas tabelas;
- **Integridade:** essa categoria define restrições de integridade que os dados contidos nas tabelas devem seguir

- **Definição de visões:** define novas maneiras de visualizar os dados, podendo também combinar várias tabelas para gerar o resultado.
- **Controle de transições:** controla o começo e o fim de cada bloco de comandos, permitindo que todos os comandos sejam descartados caso algum erro ocorra.
- **SQL dinâmico:** permite que linguagens de programação de propósito geral – Java ou C++ por exemplo, possam executar código SQL utilizando sua sintaxe nativa.
- **Autorização:** define permissões de acesso á tabelas e a visões.

A linguagem SQL permite que dados entre tabelas sejam relacionados através de operações *JOIN,* ligando os dados de diferentes tabelas, através das suas chaves estrangeiras, utilizadas para referenciar dados contidos em diferentes tabelas [26]. Apesar da simplicidade do modelo, é muito importante que sejam aplicadas técnicas de normalização, para que sejam evitados problemas como redundância de dados e desempenho nas consultas [26].

7.2 Bancos Orientados a Objeto

Segundo SILBERSCHATZ (2011), os sistemas tradicionais que manuseavam dados relativamente simples se comportam muito bem com os sistemas de bancos relacionais. Sistemas mais complexos surgiram e viu-se a necessidade de criar um novo paradigma de bancos de dados, a solução foi utilizar uma abordagem já existente para manusear esses dados – O paradigma orientado a objeto.

Os bancos de dados relacionais conseguem tratar muito bem tipos de dados primitivos tais como tipos inteiros, *String*, ponto flutuante, etc. Embora não consigam trabalhar diretamente com tipos mais complexos como estruturas, objetos, dados encadeados e dados com herança. Com isso, surgem os bancos de dados baseados no paradigma orientado a objeto [26].

O paradigma combina a definição de tabelas com a definição de objetos, fundindo os dois conceitos e criando tabelas de objetos, podendo também incluir herança entre tabelas [26].

7.3 Bancos *NoSQL*

De acordo com SULLIVAN (2015), os bancos de dados NoSQL (do inglês, non-SQL ou non-relational) emergem de um novo leque de requisitos; os sistemas estão processando

uma quantidade cada vez maior de dados, necessitam de um tempo melhor de resposta e de uma alta taxa de disponibilidade. Os bancos de dados tradicionais não estavam acompanhando essa evolução, sendo muito caro ou até impossível o processo de escalabilidade desses sistemas [28]. Foi daí que um novo paradigma surgiu – Os bancos *NoSQL*.

Esse novo paradigma não utiliza o armazenamento tradicional feito em tabelas, permitindo que esses sistemas fossem escalados horizontalmente, utilizando-se de vários *clusters* (muitas vezes de pequeno porte) [28]. Esse processo é muito difícil de ser feito com a tecnologia relacional de dados, onde a forma mais tradicional de escalar esses sistemas é vertical, ou seja, é feita adicionando mais poder de processamento e armazenamento em um único *cluster* – o que torna o processo muito caro *[28]*.

Os sistemas *NoSQL* se dividem em quatro grandes categorias:

- **Chave-valor:** é um modelo simples, onde o documento de armazenamento possui chaves e valores. Cada chave corresponde a um valor, que pode representar desde um simples valor inteiro até agregações complexas de dados.
- **Baseada em Documento:** é baseado na mesma ideia de chave-valor, mas além dos valores, também permite que sejam armazenados *meta-dados* que auxiliam a manipulação desses dados.
- **Família de colunas:** similar ao modelo relacional, a estrutura interna de armazenamento é similar as tabelas do modelo relacional, embora não possam ser aplicadas os mesmos conceitos. É ideal para grandes quantidades de dados.
- **Orientada a Grafos:** baseada na teoria dos grafos, é ideal para modelar dados que possuem um alto grau de conectividade entre si. Desde redes de computadores até sistemas de rede social.

Segundo FOWLER (2012), além da capacidade de manipular grandes quantidades de dados e poder escalar bem nos servidores, algumas empresas escolhem utilizar os bancos *NoSQL* simplesmente porque a natureza dos seus dados se adequa melhor a um dos quatro tipos de bancos *NoSQL*, tornando a produtividade do sistema muito maior do que se fosse utilizado o sistema tradicional relacional.

7.4 Bancos *NewSQL*

Os bancos de dados *NoSql* tentam resolver alguns problemas emergentes da evolução dos sistemas: armazenamento de grande quantidade de dados com alto desempenho. O que estava sendo difícil de se resolver com os bancos relacionais [32]. Para isso, os bancos NoSQL tiveram que sacrificar algumas características dos bancos tradicionais, tais como

transações, uso da linguagem SQL, falta de consistência entre outros. Os bancos *NoSQL* cresceram e se popularizaram muito nos últimos anos [32]. Embora criou-se também uma certa necessidade de se utilizar as características dos bancos tradicionais – Algo não presente no novo modelo.

O *NewSQL* veio para tentar fundir esses dois mundos – Características tradicionais, com a grande capacidade de processamento dos bancos não relacionais [31]. O novo paradigma utiliza uma estratégia um pouco diferente. Cada banco de dados se especializa no tipo de problema que tenta resolver, diferentemente do modelo tradicional, em que todo o banco é centralizado [33]. STONEBRAKER e CATTEL (2011) definem cinco características de um SGBD NewSQL:

1. Linguagem SQL como meio de interação entre o SGBD e a aplicação;
2. Suporte para transações ACID;
3. Controle de concorrência não bloqueante, para que as leituras e escritas não causem conflitos entre si;
4. Arquitetura que forneça um maior desempenho por nó de processamento;
5. Arquitetura escalável, com memória distribuída e com capacidade de funcionar em um aglomerado com um grande número de nós.

A tabela abaixo mostra um comparativo entre os principais tipos de banco de dados, podemos concluir que o *NewSQL* tenta mesclar o melhor dos dois paradigmas – *NoSQL e o* modelo relacional.

Característica	RDBMS	NoSQL	NewSQL
Cumprimento ACID (dados, integridade de transações)	Sim	Não	Sim
OLAP / OLTP	Sim	Não	Sim
A análise dos dados (agregados, transformar, etc.)	Sim	Não	Sim
Rigidez do esquema (mapeamento rigoroso da modelo)	Sim	Não	Talvez
Flexibilidade formato de dados	Não	Sim	Talvez
A computação distribuída	Sim	Sim	Sim
Escala para cima (vertical) / Dimensionar (horizontal)	Sim	Sim	Sim
Desempenho com crescimento de dados	Rápido	Rápido	Muito Rápido
Sobrecarga de desempenho	Enorme	Moderado	Mínimo
Popularidade / Suporte comunidade	Enorme	Crescente	Crescendo lentamente

Figura 18: Comparativo entre os principais paradigmas de banco de dados
Fonte: http://www.devmedia.com.br/conheca-a-geracao-de-banco-de-dados-nosql-e-newsql/33202. Acesso em: 11 set 2017

3. O sistema

Esse capítulo fornece uma visão do sistema por meio de engenharia de requisitos. São listados os principais requisitos funcionais e não funcionais; casos de uso e diagramas de classe.

3.1 Visão Geral

O sistema é composto por dois subsistemas, um do tipo *ERP (*do inglês, *enterprise resource planning)* e outro do tipo *POS* (do inglês, *point of sales*). Os sistemas controlaram todos os processos da empresa, desde a venda até a entrega do produto final.

A empresa trabalha com confecção de roupas de luxo sob medida. Ela é composta por vários setores – vendas, controle de qualidade, produção, acabamento, logística e etc. Cada setor possui processos bem definidos que devem funcionar por meio dos sistemas. Basicamente o setor de vendas irá coletar informações para que a roupa seja confeccionada, tais como: medidas corporais dos clientes, design escolhido de cada roupa, tipo de corpo e possíveis imperfeições de cada cliente, entre outros dados.

Depois da produção, existem processos de logística e controle de qualidade. A roupa pode ser fabricada em diferentes regiões do mundo, dependendo de vários fatores, um dos principais sendo a disponibilidade e custos de material bruto para confecção e de mão de obra. Com isso, é necessário enviar os produtos para diferentes localizações, para isso, o sistema possui um setor de logística. Além disso, para garantir a qualidade dos produtos, a empresa aplica processos de controle de qualidade, em que o cliente deverá provar a roupa e o estilista responsável pela venda identificará possíveis erros. Caso haja alguma imperfeição, os alfaiates deverão efetuar alterações na roupa, até que ela sirva perfeitamente ao cliente, somente depois disso, o ciclo básico de atividade da empresa é concluído.

3.2 Requisitos Funcionais

[RF01] Receber submissão de pedidos do sistema POS

Descrição	O sistema deverá ser capaz de receber pedidos vindo do sistema POS, utilizado pelos vendedores. O ERP deve então processar a requisição e retornar uma mensagem de erro ou sucesso para o POS.
Casos de uso relacionados	[UC01] – Submissão de venda

[RF02] Atualização de medidas dos clientes

Descrição	O sistema permitirá ao administrador que atualize as medidas corporais de cada cliente.
Casos de uso relacionados	[UC04] – Atualizar medidas

[RF03] Atualização de design do produto

Descrição	O sistema permitirá que o administrador altere o design de cada roupa
Casos de uso relacionados	[UC05] – Atualizar design

[RF04] Atualização do padrão corporal de cada cliente

Descrição	O sistema permitirá que o administrador atualize o padrão corporal de cada cliente.
Casos de uso relacionados	[UC06] – Atualizar o padrão corporal

[RF05] Geração de PDF para produção

Descrição	O sistema permitirá aos usuários administradores e de controle de produção que seja gerado um PDF com todos os detalhes necessários para que sejam produzidos os produtos. Os PDF devem estar disponíveis em inglês e também na língua nativa do país de produção do produto.
Casos de uso relacionados	[UC08] – Geração de PDF para produção [UC09] – Tradução de PDF de produção

[RF06] Organização de logística

Descrição	O sistema permitirá aos usuários administradores e de controle de produção, que organizem os produtos em caixas e definam a data e o destino de cada caixa, para que os produtos possam chegar a outros destinos.
Casos de uso relacionados	[UC07] – Organização de logística

[RF07] Agendamento de provas

Descrição	O sistema permitirá aos administradores e vendedores que sejam agendados encontros com os clientes, para que se defina se a roupa serve perfeitamente ou se são necessárias correções na roupa.
Casos de uso relacionados	[UC03] – Agendamento de prova

[RF08] Processo de correção

Descrição	O sistema permitirá que os administradores encaminhem produtos para que sejam feitas correções. As correções podem ocorrer em diferentes cidades com diferentes alfaiates.
Casos de uso relacionados	[UC11] – Processo de correção de roupas

[RF09] Recebimento de pagamentos

Descrição	O sistema permitirá que usuários vendedores e administradores recebam pagamentos dos clientes. Os pagamentos podem ser recebidos via *endpoint* através do sistema POS ou diretamente no sistema ERP.
Casos de uso relacionados	[UC02] – Processar Pagamentos

[RF10] Finalizar pedido

Descrição	O sistema permitirá que os administradores concluam um pedido, uma vez que todos os processos tenham sido concluídos com êxito.
Casos de uso relacionados	[UC13] – Finalizar pedido

[RF11] Tradução de PDFs

Descrição	O sistema permitirá que os usuários de produção possam traduzir comentários e instruções escritas em inglês para a língua nativa do país onde a roupa será fabricada.
Casos de uso relacionados	[UC09] – Tradução de PDF de produção

[RF12] Atualização da forma corporal de cada cliente

Descrição	Após o fim de cada pedido, o sistema permitirá que os usuários responsáveis pela produção possam atualizar a forma corporal (localizada fisicamente na fábrica em que a roupa foi produzida) de cada cliente.
Casos de uso relacionados	[UC12] – Atualizar forma corporal

[RF13] Notificação de erros de produção

Descrição	O sistema permitirá que usuários de produção notifiquem a sede da empresa de qualquer tipo de problema ou dúvidas durante a produção do produto.
Casos de uso relacionados	[UC10] – Notificar erros de produção

3.3 Requisitos Não Funcionais

Sistema

[RNF01] Especificação de Software

Descrição	O sistema deverá ser desenvolvido em linguagem de programação PHP versão 5.6.10
Casos de uso relacionados	Todos

[RNF02] Banco de Dados

Descrição	O sistema fará uso do banco de dados relacional MySQL versão 5.5.57-cll para Linux RedHat, para manter os dados do sistema.
Casos de uso relacionados	Todos

[RNF03] Servidor

Descrição	O sistema deverá funcionar no servidor Apache/2.2.31 utilizando *FastCGI* e compressão *Gzip*, além de suportar os protocolo HTTP/1.1
Casos de uso relacionados	Todos

[RNF04] Versão da biblioteca *Jquery UI*

Descrição	O sistema deve utilizar a versão 1.11.4
Casos de uso relacionados	Todos

[RNF05] Versão da biblioteca *Bootstrap*

Descrição	O sistema deve utilizar a versão 2.1.1
Casos de uso relacionados	Todos

RNF06] Versão da biblioteca *Raphael.js*

Descrição	O sistema deve utilizar a versão 2.0.1
Casos de uso relacionados	Todos relacionados aos processos de alteração

RNF07] Versão da biblioteca *Tagdd.js*

Descrição	O sistema deve utilizar a versão 2.0
Casos de uso relacionados	Todos relacionados com manipulação de imagens

RNF08] Versão da biblioteca FFPDF

Descrição	O sistema deve utilizar a versão 1.6
Casos de uso relacionados	Todos relacionados com geração de PDFs

RNF08] Versão da biblioteca FFMPEG

Descrição	O sistema deve utilizar a versão 0.10.16
Casos de uso relacionados	Todos relacionados com geração de PDFs

RNF09] Versão da biblioteca *GroceryCrud*

Descrição	O sistema deve utilizar a versão 1.5.4
Casos de uso relacionados	Todos

Segurança

[RNF10] Recebimento de dados via *End-point*

Descrição	Todas as transações de dados feitas com o sistema *POS* devem passar por autenticação primeiro. A autenticação deve verificar nome de usuário e senha. Os dados de senha devem vim do POS através de uma requisição HTTPS utilizando certificados TSL/SSL.
Casos de uso relacionados	Todos que envolvam autenticação de usuários

[RNF11] Comunicação segura entre cliente e servidor

Descrição	O sistema deve suportar o protocolo HTTPS (versão x.x) e utilizar certificados no lado do servidor do tipo SSL e TSL, mais especificamente o *OpenSSL*. Implementando os algoritmos PKCS SHA-256 com criptografia RSA
Casos de uso relacionados	Todos que envolvam autenticação de usuários

[RNF12] Backup de Dados

Descrição	O sistema fará um *backup parcial* do seu banco de dados todos os dias e guardará os dados por 1 mês.
Casos de uso relacionados	-

[RNF13] Dados sensíveis

Descrição	Todos os dados sensíveis do sistema tais como: senhas ou dados importantes de clientes devem ser armazenados de forma

	criptografada com o algoritmo *BCrypt*.
Casos de uso relacionados	-

[RNF14] Comunicação entre sistemas

Descrição	Os sistemas ERP e POS devem se comunicar utilizando o protocolo HTTP POST. Respostas a cada requisição devem ser informadas. Em caso de erro os sistemas também devem indicar isso na resposta.
Casos de uso relacionados	[UC01] – Submissão de venda [UC02] – Processar Pagamentos [UC03] – Agendamento de prova

Portabilidade

[RNF15] Dispositivos suportados

Descrição	O usuário poderá acessar o sistema em dispositivos Desktop com acesso à Internet, através dos *browsers* Mozilla Firefox 20+ com JavaScript ativado.
Casos de uso relacionados	Todos

Internacionalização

[RNF16] Idioma

Descrição	O sistema deverá apresentar a interface com o usuário na língua inglesa. E também deve ser capaz de gerar todos os arquivos de produção dos produtos (PDfs de produção e instruções dentro do sistema) na língua nativa do país onde a fábrica está localizada.
Casos de uso relacionados	Todos

3.4 Casos de uso

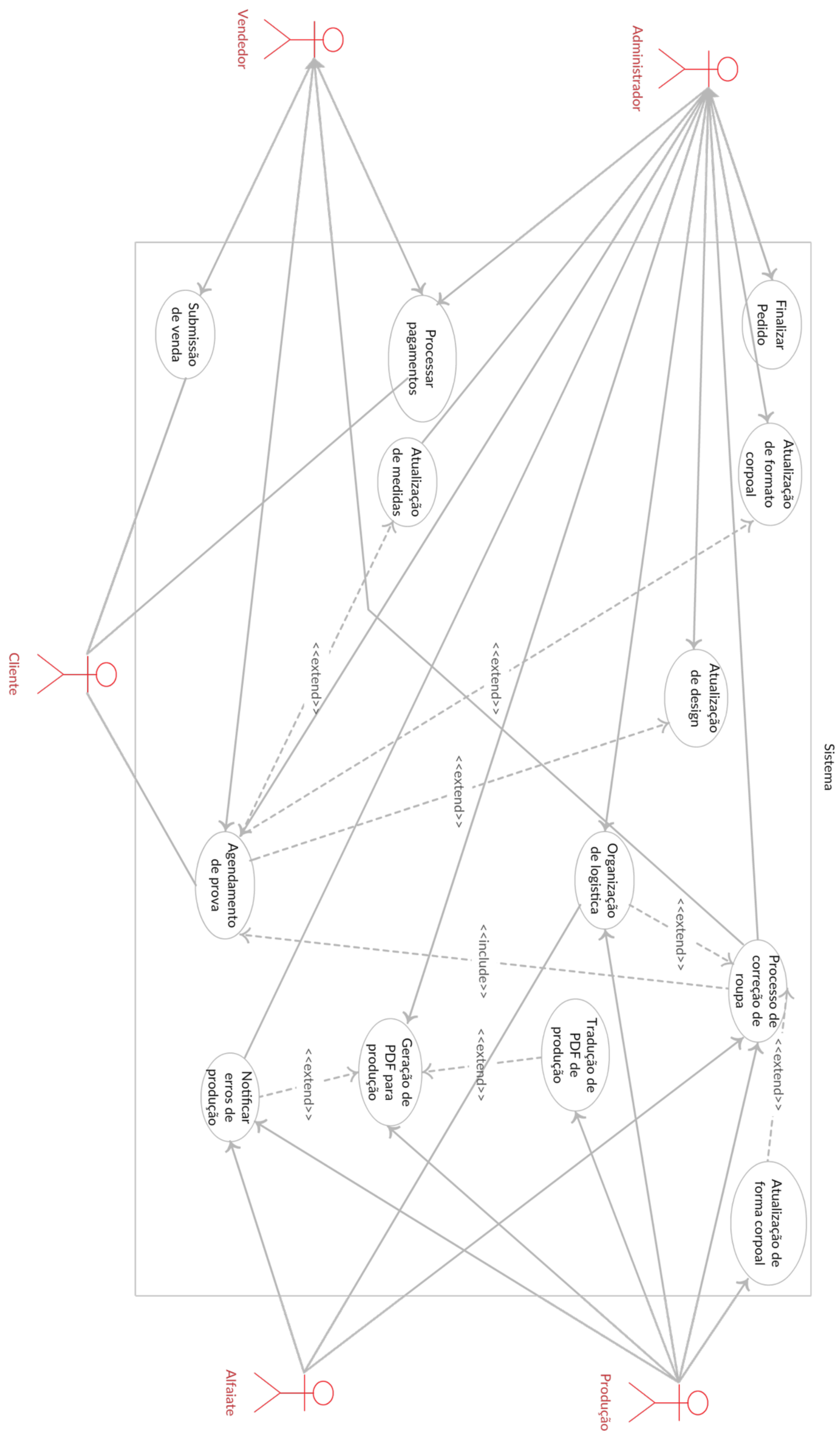

Figura 19: Diagrama UML de caso de uso

3.4.1 [UC01] – Submissão de venda

Descrição	Processo inicial de venda e aquisição de informações dos clientes, o processo deve ser efetuado com muito cuidado, uma vez que os dados serão enviados para a sede da empresa e a produção dos produtos será baseada nesses dados.
Ator	Vendedor e Cliente
Pré-condições	1. Existir Cliente cadastrado no sistema
Pós-condições	Pedido estará submete e disponível para acesso e modificações no sistema ERP.
Fluxo Principal	1. Realizar a captura de informações do pedido 2. Vendedor submete pedido para o ERP
Fluxo Secundário	**- Cliente não existente** Caso o cliente não esteja cadastrado ainda, o usuário deve submeter as informações do novo cliente primeiro.

3.4.2 [UC02] – Processar Pagamentos

Descrição	Clientes poderão efetuar pagamentos referentes aos seus pedidos
Ator	Vendedor, Administrador e Cliente
Pré-condições	Necessário que o cliente já possua ordem não finalizada dentro do sistema
Pós-condições	Saldo devedor do cliente será reduzido ou zerado.
Fluxo Principal	1. Clientes submetem pagamentos pelo sistema POS juntamente com os vendedores 2. Pagamento é enviado ao ERP através de *end-points*
Fluxo Secundário	**- Vendedor não presente** Clientes submetem pagamentos pelo sistema ERP juntamente com os administradores.

3.4.3 [UC03] – Agendamento de prova

Descrição	Vendedores e Administradores agendam sessão para que os clientes possam provar os produtos já manufaturados. O processo de prova é necessário para que se identifique possíveis erros ou imperfeições no produto.
Ator	Vendedor, Administrador e Cliente
Pré-condições	Necessário que os produtos dos clientes já estejam manufaturados e prontos para serem provados
Pós-condições	Ordem é finalizada ou produto é redirecionado para alterações de correção
Fluxo Principal	1. Vendedores ou administradores contatam os clientes e marcam uma sessão no dia e hora convenientes para os dois. 2. Clientes provam e avaliam o produto final juntamente com o vendedor 2.1 [Extends [UC11] – Processo de correção de roupas] 2.2 [Extends [UC04] – Atualizar medidas] 2.3 [Extends [UC05] – Atualizar design] 2.4 [Extends [UC06] – Atualizar o padrão corporal] 3. Ordem finalizada
Fluxo Secundário	

3.4.4 [UC04] – Atualizar medidas

Descrição	Administradores podem atualizar valores de medidas corporais dos clientes para que os dados gerados de produção sejam atualizados.
Ator	Administradores
Pré-condições	Necessário que o cliente já possua cadastro de medidas no sistema
Pós-condições	Próximos PDFs de produção serão gerados com os valores mais atuais de medidas. E os valores antigos deverão ir para o histórico de medidas dos clientes
Fluxo Principal	1. No menu de dados de cada cliente, administradores deverão inserir novas medidas

	2. Devem salvar suas alterações
Fluxo Secundário	-

3.4.5 [UC05] – Atualizar Design

Descrição	Administradores podem atualizar o design de cada produto dos clientes, para que os produtos possam ser produzidos corretamente.
Ator	Administradores
Pré-condições	Necessário que o cliente já possua pelo menos uma ordem não finalizada e não em produção
Pós-condições	Próximos PDFs de produção serão gerados com os valores mais atuais de design. E os valores antigos deverão ir para o histórico de design dos clientes
Fluxo Principal	1. No menu de dados de cada ordem, administradores devem escolher o produto desejado 2. Depois devem clicar em "Design" e serão levados a página de design do produto. 3. Após feitas as alterações os administradores deverão salvar os dados
Fluxo Secundário	-

3.4.6 [UC06] – Atualizar padrão corporal

Descrição	Administradores podem atualizar tipo de de formato corporal de cada cliente, com isso, os PDFs de produção terão informações mais precisas para a produção dos produtos.
Ator	Administradores
Pré-condições	Necessário que o cliente já possua cadastro de formato corporal no sistema
Pós-condições	Próximos PDFs de produção serão gerados com os valores mais

	atuais de formato corporal. E os valores antigos deverão ir para o histórico de formatos corporais
Fluxo Principal	1. Dentro dos clientes, os administradores deverão clicar na opção de editar formato corporal 2. Após efetuar as mudanças devem clicar em Salvar.
Fluxo Secundário	-

3.4.7 [UC07] – Organização de logística

Descrição	Administradores e fabricantes podem organizar como deve ser feito os processos de logística de cada produto, quando é necessário que certos processos (produção, ajustes) sejam efetuados em outros locais.
Ator	Administradores, fabricantes e alfaiates
Pré-condições	Necessário que exista um produto a ser entregue em outra localização
Pós-condições	Metadados de localização de cada produto serão atualizados dentro do sistema.
Fluxo Principal	1. No menu de administradores, o usuário deve clicar em *Manage shipping boxes (*do inglês, Organizar logística de produtos *)* 2. Os usuários devem arrastar cada produto para suas determinadas caixas, caso não exista uma caixa adequada (com data de envio e destino requerido), o usuário pode criar novas caixas. 3. Após enviar fisicamente a caixa, o usuário deve indicar isso no sistema e preencher mais dados, tais como : custo de envio e código de rastreamento.
Fluxo Secundário	1. No menu de fabricantes, o usuário deve clicar em *Manage shipping boxes (*do inglês, Organizar logística de produtos *)* 2. Os usuários devem arrastar cada produto para suas determinadas caixas, caso não exista uma caixa adequada (com data de envio e destino requerido), o usuário pode criar novas caixas. 3. Após enviar fisicamente a caixa, o usuário deve indicar isso no sistema e preencher mais dados, tais como: custo

	de envio e código de rastreamento.

3.4.8 [UC08] – Geração de PDF para produção

Descrição	Administradores e fabricantes podem gerar PDF para produção. Esse PDF conterá todas as informações necessárias para que os produtos sejam manufaturados.
Ator	Administradores e fabricantes
Pré-condições	Necessário que o cliente já possua uma ordem que ainda não está em produção
Pós-condições	Após gerado o PDF e a ordem for aprovada para produção, os fabricantes poderam encaminhar os PDFs para que os produtos sejam manufaturados.
Fluxo Principal	1. Dentro dos detalhes de cada produto, o Administrador deve aprovar todos os dados – medidas, design, formato corporal e data de entrega e então o pdf será gerado em inglês. 2. Fabricante deve verificar se existe algo a ser traduzido para a língua nativa da fabrica. 3. 3.2 [Extends [UC09] – Tradução de PDF de produção] 4. Depois os fabricantes terão que gerar o PDF na língua nativa e só então, efetuar a produção do produto.
Fluxo Secundário	**- Impossibilidade de produção** Caso exista alguma informação não clara ou errada, o fabricante deverá notificar os administradores através da criação de erros dentro do sistema e só depois a produção deverá continuar normalmente

3.4.9 [UC09] – Tradução de PDF de produção

Descrição	Fabricantes devem traduzir informações essenciais para a produção do produto na língua nativa da fábrica, uma vez que um PDF na língua nativa dos fabricantes resulta em um produto final com mais qualidade.
Ator	Fabricantes
Pré-condições	Necessário que o PDF em inglês já tenha sido aprovado pelos administradores
Pós-condições	Um PDF na língua solicitada, pronto para ser utilizado na produção do produto deve ser gerado
Fluxo Principal	1. Dentro dos detalhes de cada produto, os administradores devem traduzir todos os campos essenciais para a produção. Os campos serão indicados pelo sistema e será fornecida uma interface de tradução adequada para cada campo automaticamente. 2. Após todos os campos forem traduzidos, a geração de PDF será liberada na língua desejada e a produção poderá começar.
Fluxo Secundário	-

3.4.10 [UC10] – Notificar erros de produção

Descrição	Fabricantes podem notificar administradores de erros ou dúvidas durante os processos de produção.
Ator	Fabricantes e Administradores
Pré-condições	Necessário que o produto já esteja em processo de produção
Pós-condições	*Tickets* serão criados no sistema para manter histórico de erros.
Fluxo Principal	1. Durante a produção, os fabricantes podem notar alguma inconsistência nos dados – Medidas dos clientes, design da roupa, formato corporal e outros. 2. Caso algo impeça a produção do produto, ele deve criar um *ticket* dentro da janela de dados sobre a ordem. 3. Administradores devem ser notificados e respondem imediatamente para que a produção possa ser continuada. Em casos mais graves, o cliente deve ser contatado para esclarecer qualquer problema.

Fluxo Secundário	-

3.4.11 [UC11] – Processo de correção de roupas

Descrição	Após uma prova, podem ser encontrados problemas em cada produto, esses problemas devem ser corrigidos no processo de correção.
Ator	Alfaiates, fabricantes, vendedores e administradores
Pré-condições	Necessário que existam produtos não perfeitos já completamente manufaturados
Pós-condições	O produto final terá correções efetuadas estará pronto para outro processo de prova.
Fluxo Principal	1. Caso o produto não sirva perfeitamente, Vendedores ou administradores criarão um processo de alteração no menu principal do sistema. Deve ser informado qual produto será alterado, o que deve ser alterado, o alfaiate responsável pela alteração e a data limite. 2. Depois, deve-se encaminhar os produtos para o alfaiate designado, podendo ser necessário utilizar a seção de logística para isso. 2.1 [Extends [UC07] – Organização de logística] 3. Após o recebimento do produto, o alfaiate terá que confirmar os dados no sistema, efetuar as alterações e submeter um relatório do que foi feito. 4. Depois ele terá que enviar de volta o produto para a sua origem e um novo processo de prova será efetuado. 4.1 [Include [UC03] – Agendamento de prova] 5. Caso a alteração seja muito comprometedora, o formato corporal do cliente deve ser atualizado 5.1 [Extends [UC12] – Atualizar forma corporal]
Fluxo Secundário	-

3.4.12 [UC12] – Atualizar forma corporal

Descrição	Fabricantes podem atualizar fisicamente as formas corporais usadas na produção de cada produto. As formas são placas, que servem de molde para os produtos. Isso é muito importante para que se tenha um guia de como fabricar os produtos para cada

	cliente.
Ator	Fabricantes
Pré-condições	Necessário que já exista informação de um produto que sirva bem em cada cliente.
Pós-condições	A forma corporal localizada na fábrica será alterada e os próximos produtos feitos para o cliente terão mais qualidade e precisão.
Fluxo Principal	1. Após conseguir fabricar um produto que sirva bem aos clientes ou após alterações de correção do produto, os fabricantes devem ser notificados para que atualizem as formas corporais. 2. Na fábrica, os fabricantes irão alterar fisicamente a forma e indicar o dia, data e quais mudanças foram efetuadas.
Fluxo Secundário	-

3.4.13 [UC13] – Finalizar pedido

Descrição	Administradores devem finalizar pedidos. Isso significa que todas as pendencias com o cliente estão concluídas.
Ator	Administradores
Pré-condições	Necessário que todas as condições sejam atendidas: Cliente esteja com os produtos, todos os pagamentos tenham sido efetuados e nenhuma pendência exista com os fabricantes e alfaiates.
Pós-condições	Informações sobre determinado pedido deixará de existir nos processos principais do sistema e será arquivado.
Fluxo Principal	6. Dentro da tela de informações de cada pedido os administradores deverão clicar no botão "Finalize Order". 7. O sistema então verificará se algumas pendências tenham sido alcançadas, tal como saldo devedor dos clientes e localização de cada produto. 8. Se tudo estiver nos conformes a ordem será arquivada, se não mensagens de erro serão lançadas no sistema.
Fluxo Secundário	-

3.5 Arquitetura

A arquitetura do sistema é baseada no modelo *MVC* (do inglês, *Model-View-Controller*) e na Figura 20 é descrita uma visão de alto nível da mesma que utiliza como base o framework PHP Codeigniter. A camada de visão do sistema deve interagir diretamente com os usuários, captando entradas e exibindo resultados. Através de requisições HTTP os usuários se comunicam com a camada dos controladores no sistema. Essa camada, então, utiliza diversos modelos para executar a lógica de negocio e retornar os resultados para os usuários. A camada de modelos executa toda a lógica de negocio do sistema e também manipula o banco de dados.

Algumas bibliotecas auxiliares também são utilizadas no sistema, são elas: Jquery, JqueryUI, bootstrap, tagdd, raphael.js, FPDF, FFMPEG e GroceryCRUD (ver Figura 20). A camada Controlador do sistema, serve de end-point para outro sistema – POS. Este utiliza a API fornecida pelo ERP para que os dois sistemas se comuniquem. O usuário do ERP deve utilizar requisições HTTP para se comunicar e utilizar o sistema. A camada de visão deverá disparar essas requisições que serão recebidas pelos controladores que por sua vez utilizarão os modelos e bibliotecas auxiliares para formar a resposta e retorná-las para a camada de visão.

A biblioteca Jquery é uma das maiores bibliotecas Javascript existentes. Ela abstrai várias funcionalidades nativas do Javascript, tais como: seleção e manipulação de elementos visuais e tratamento de eventos e animações. Além disso, a biblioteca permite que o mesmo código funcione da mesma forma nos navegadores mais populares, já que muitas vezes existe incompatibilidades entre navegadores [38].

A biblioteca JqueryUI possui uma grande variedade de elementos visuais e efeitos que podem ser aplicados facilmente nos sistemas. Com ela, é possível criar facilmente elementos gráficos simples e avançados em uma página Web. A biblioteca utiliza como base a biblioteca Jquery, então é necessário que as duas sejam incluídas no projeto [39].

Bootstrap é um dos mais famosos e completos *frameworks front-end* do mercado. O framework contém elementos já prontos, desde simples botões até os mais complexos módulos para um sistema *web*. O framework também fornece elementos prontos para serem utilizados por diferentes dispositivos, permitindo que o sistema tenha consistência em praticamente qualquer ponto de acesso [40].

A biblioteca *Taggd.js* é uma ferramenta muito simples que serve para marcar elementos em imagens. Com ela é possível inserir etiquetas e a sua descrição com posições X e Y na imagem. A biblioteca é utilizada em diversas partes do sistema, já que seria praticamente impossível realizar todos os processos de fabricação e correções dos produtos sem auxilio visual de imagens [41].

A biblioteca *RaphaelJS* é uma biblioteca leve que permite ao usuário utilizar *SGV* (*do inglês, Scalable Vector Graphic*) diretamente no navegador. Ela é importante pois permite que alfaiates assinalem exatamente quais alterações foram feitas nos painéis corporais dos clientes através de desenhos de vetores [42].

A biblioteca *FFPDF* é responsável por gerar arquivos no formato PDF, esses arquivos são utilizados para auxiliar a produção do produto, descrever instruções de alteração de roupas e geração de relatórios. A biblioteca é codificada em PHP e é executada pelos controladores e modelos do sistema [43].

O FFMPEG é na verdade um software que possui diversas bibliotecas para manipulação de vídeo e áudio. Ela é importante no sistema, pois permite que os vídeos e imagens sejam de forma muito meticulosa, permitindo a extração de frames específicos, rotação de imagens e etc. [44].

O *GroceryCRUD* é uma biblioteca escrita em PHP que permite a geração de código automático para as ações básicas de uma entidade – criação, remoção e atualização. Essa ferramenta se encarrega de gerar visões, controladores e modelos automaticamente [45].

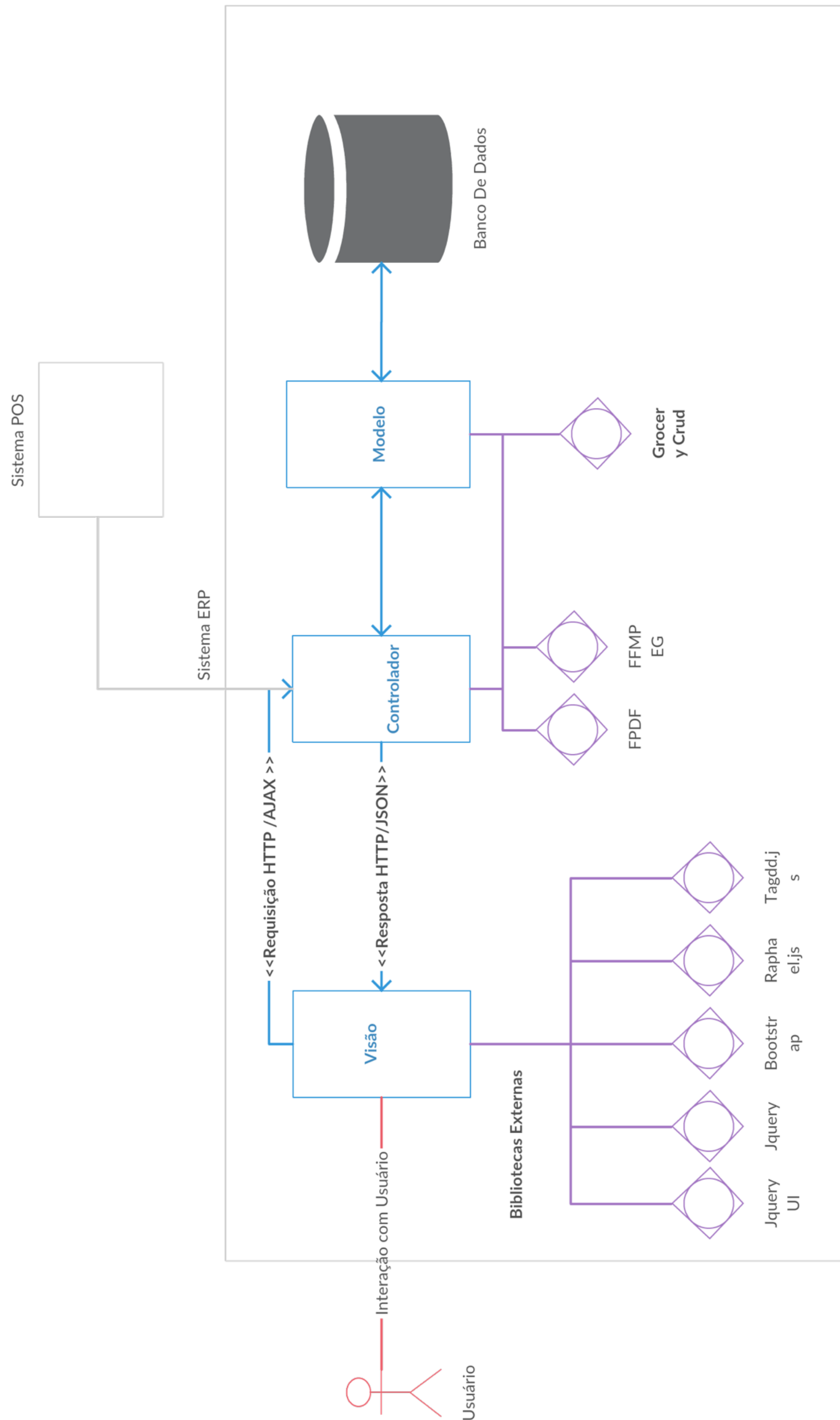

Figura 20: Arquitetura de alto-nível do sistema

3.5.1 Arquitetura de baixo Nível

O sistema trabalha com uma arquitetura MVC e um fluxo de trabalho pré-defindo deve ser seguido rigorosamente (figura 21). As classes controladoras devem ser intermediadoras entre as camadas de visão e os modelos. Dessa forma, o framework encoraja o usuário a não utilizar lógica de negócios dentro dos controladores, sabendo disso, o framework restringe a chamada de controladores por qualquer entidade da camada de modelos ou outros controladores, somente sendo acessíveis através de requisições HTTP ou por linha de comando. Por outro lado, os modelos são capazes de instanciar outros modelos assim como os controladores; e devem ser utilizados para realizar a lógica de negócio.

Os modelos são classes que devem conter a lógica de negócio do sistema e também manipulam o banco de dados através de padrão de projeto fornecido pelo *Codeigniter* chamada de *Object Relational Mapping (ORM)*. Esse padrão abstrai o acesso ao banco de dados, fazendo-se muito fácil e rápido o desenvolvimento da aplicação. O uso do *ORM* no *Codeigniter* permite que diferentes tipos de bancos de dados sejam utilizados sem a necessidade de alterar o código de acesso. A técnica também permite que os objetos contidos no banco de dados sejam criados e mantidos na memória no momento em que a requisição é feita, removendo assim a necessidade de se ter classes bem definidas representando objetos estáticos na aplicação, permitindo uma maior flexibilidade em consultas no banco de dados [37].

As classes da camada de visão representam o sistema graficamente para o usuário e realizam chamadas para os controladores do sistema. Essas chamadas são responsáveis pela navegação do sistema e por realizar ações no sistema, por exemplo: mudar de página, mostrar detalhes de pedidos específicos, mostrar listas específicas, aprovar pedidos para produção, atualizar opções de design, entre outros. Depois de disparar essas ações no sistema, os controladores devem então tratar cada requisição, utilizando a camada dos modelos, que por sua vez, utilizam bibliotecas e o banco de dados para efetuar essas ações.

É importante ressaltar que o framework leva o programador a manter fracas relações entre os componentes da camada de controladores, em outras palavras os controladores não devem "conhecer" outros controladores, simplificando muito o desenvolvimento, já que o programador deve seguir à risca o fluxo de execução proposto pelo framework. Por outro lado, o framework encoraja o programador a criar relações entre os modelos, cada modelo pode manter relações com outros modelos, isso permite que a lógica de negócio fique bem

definida e separada em cada arquivo, evitando repetição de código e ao mesmo tempo permitindo que o a lógica contida em um modelo seja reaproveitada em outros.

Por exemplo, o modelo responsável pelos pedidos (*Order model*) realiza diversas operações, essas operações muitas vezes utilizam dados dos produtos de cada pedido, dados dos clientes e dos pagamentos, por exemplo. Dessa forma, o modelo de pedidos possui chamadas para diversos outros modelos. Vale salientar também que todos os modelos possuem uma referência para o *ORM* do Codeigniter, através do padrão *Singleton*. Assim, diferentes chamadas ao banco de dados podem ser feitas pela mesma instância do ORM, mesmo sendo chamados de diferentes modelos, dessa forma o desempenho não é muito prejudicado caso a requisição instancie diversos modelos.

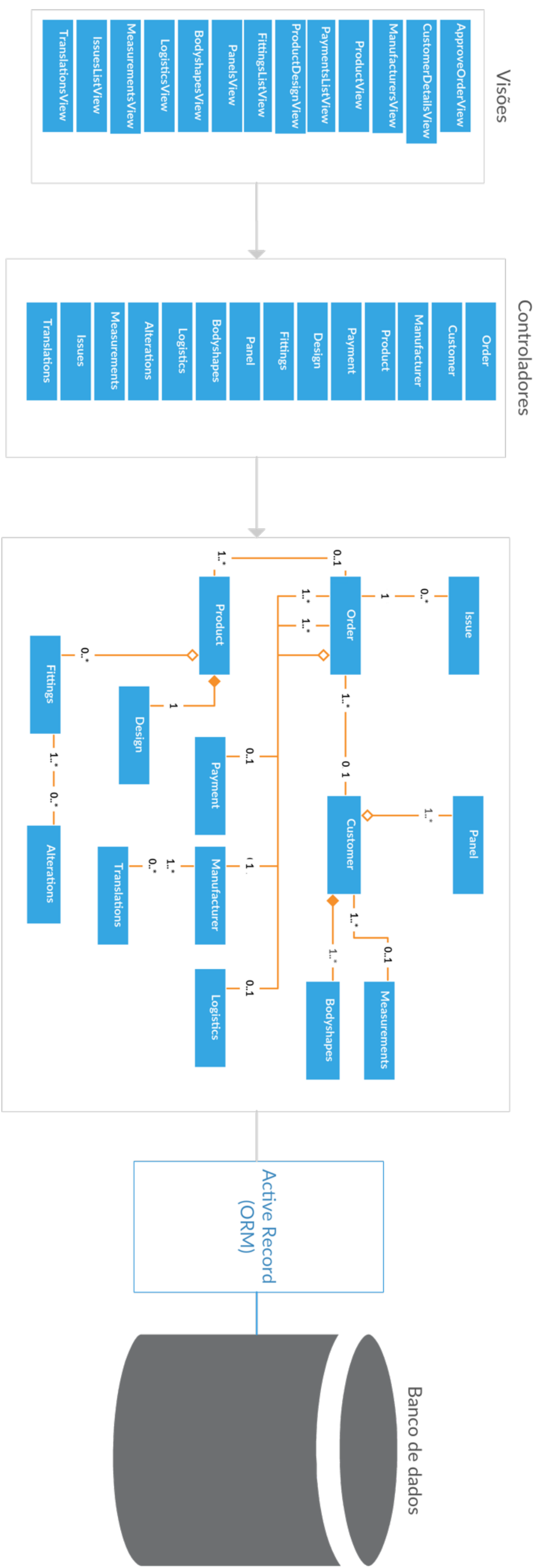

Figura 21: Arquitetura de baixo nível

4. Avaliação e proposta de evolução da arquitetura

Esse capítulo descreve alguns problemas encontrados na arquitetura atual do sistema, também, a descrição do fluxo de execução atual do sistema ERP, uma visão de como os processos de negócio devem funcionar e ao final do capítulo uma proposta de evolução do fluxo de execução utilizando uma nova arquitetura de sistema.

4.1 Contexto

Os processos realizados no sistema de informação da empresa são cruciais para o correto andamento da produção, acompanhamento e entrega final dos produtos. Sem ele, seria muito difícil identificar e tratar todos os aspectos de negócio desses produtos em tempo hábil, causando uma redução drástica na produção da empresa.

As atividades exercidas pelos administradores no acompanhamento dos pedidos evitam muitos erros de produção e melhoram a qualidade final dos produtos. Os administradores devem estudar meticulosamente cada pedido, a fim de identificar e corrigir possíveis erros. Esses erros podem causar grandes prejuízos a empresa, levando em conta que os itens produzidos possuem alto custo atrelado.

Um dos grandes problemas da empresa hoje é o tempo que se leva para que os administradores analisem e aprovem cada pedido, já que inúmeros fatores devem ser levados em consideração. Atualmente, cada administrador tem a capacidade de atender aproximadamente dois pedidos por hora, o que já limita muito o número de pedidos, muitas vezes forçando os vendedores a recusar novos clientes.

O sistema carece de atributos geralmente presentes em sistemas desktop, tais como boa taxa de resposta, fluidez e boa usabilidade. Esses atributos podem ser alcançados em sistemas *Web* com uma arquitetura conhecida como *SPA (*do inglês, *Single page applications) [47]*. Existem diversos frameworks baseados em Javascript que fornecem essa arquitetura, podemos citar React.JS, Angular.JS, Meteor.JS, entre outros [47]. O uso dessa arquitetura na camada de visão, possivelmente tornará usabilidade mais fluída e rápida [47], permitindo que os usuários trabalhem mais confortavelmente e consigam aprovar mais pedidos.

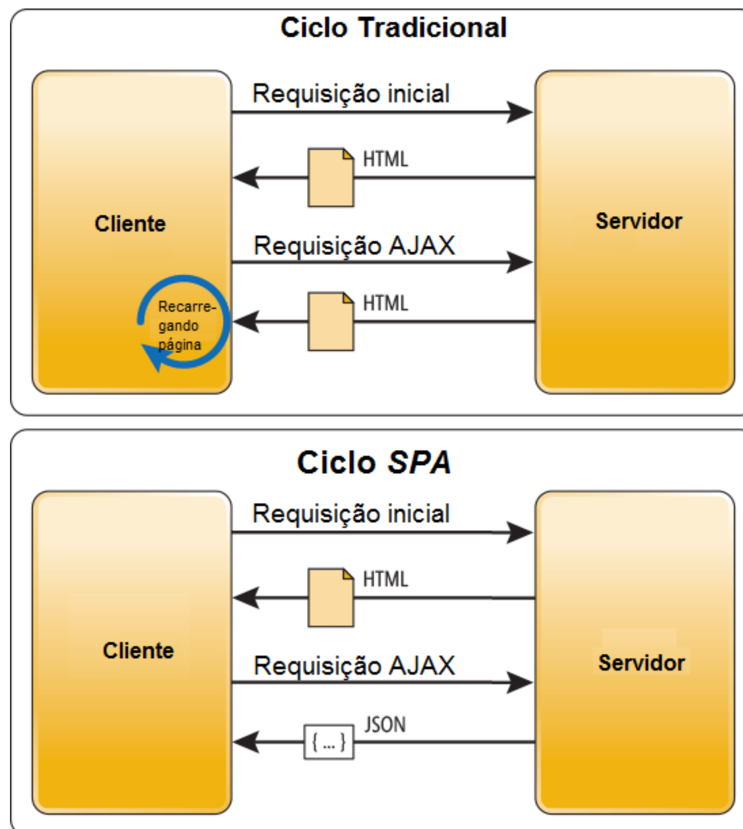

Figura 22: Comparação entre sistemas tradicionais e SPA
Fonte: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx. Acesso em: 8 nov. 2017

A solução proposta é migrar a arquitetura atual para uma arquitetura que utilize o modelo *SPA*. Para isso, será necessário que a camada de visão do sistema seja adaptada para que se consiga utilizar algum framework SPA e que se façam algumas mudanças nos controladores e modelos, embora sejam mais sutis do que as necessárias na camada de visão, visto que grande parte do código contido nos controladores e modelos poderá ser aproveitada.

4.1 Visão geral do processo de negócio

Será descrito nesta seção um dos principais fluxos de atividade do sistema – A análise e aprovação de um pedido para produção. Essa atividade é de extrema importância para que todo o resto dos processos funcionem corretamente, dado que é nessa etapa que as informações dos pedidos, inicialmente capturados pelos vendedores, serão conferidas por um administrador especialista, capaz de identificar possíveis erros ou inconsistências nos dados da compra.

Um dos principais problemas nessa etapa é a quantidade de tempo necessária pelo administrador para aprovar cada pedido. Isso ocorre pelo fato de que existem mais vendedores que administradores, fazendo com que poucos administradores tenham que aprovar muitos pedidos, muitas vezes resultando na rejeição de novos pedidos pela empresa.

O processo de aprovação de um pedido deve ser efetuado cautelosamente pelo usuário administrador e envolve muitas seções do sistema e grande processamento de dados para se gerar os PDF's de produção, e também atualizar os dados do pedido na seção de produção. O fluxo de atividade é descrito na Figura 23:

1. **Aquisição de dados:** processo inicial de captura da venda pelos vendedores. É nesse processo que todos os dados dos clientes serão capturados – imagens, vídeos, medidas, formato corporal e design da roupa.
2. **Armazenamento dos dados:** processamento do pedido e inserção dos dados, agora disponíveis para os administradores.
3. **Recorte de frames dos vídeos:** nessa etapa os administradores devem analisar e os vídeos submetidos pelos vendedores e capturar *frames* importantes do cliente. Esses *frames* ajudarão a equipe de produção a manufaturar o produto.
4. **Adição de comentários:** os administradores devem adicionar comentários explicativos em cada uma das imagens e também nos vídeos. No caso de imagens, os administradores podem colocar comentários em locais específicos da imagem, auxiliando também na produção do produto.
5. **Medidas corporais:** nessa etapa, os administradores devem analisar e possivelmente adicionar mais medidas aos clientes. Geralmente essas medidas só podem ser adicionas por usuários muito experientes.
6. **Ajustes finos no formato corporal:** nessa etapa os administradores devem verificar os detalhes submetidos pelos vendedores referentes ao formato corporal e possivelmente adicionar mais detalhes.
7. **Análise do design:** os administradores devem analisar cada opção de design submetida e identificar possíveis conflitos no design, levando em consideração todas as outras opções de design, medidas dos clientes, imagens e formato corporal. Caso alguma irregularidade seja encontrada, os administradores devem notificar os vendedores e as informações devem ser corrigidas

imediatamente (Fluxo volta para etapa 1), para que não se atrase a produção do produto.

8. **Verificação geral de inconsistências:** etapa final de checagem, onde o administrador fará uma busca geral pelos principais pontos do pedido, se tudo ocorrer bem, ele avança para o próximo passo, se não ele terá que corrigir dados incorretos com o vendedor (Fluxo volta para etapa 1).

9. **Encomenda do tecido:** última etapa antes de se iniciar a produção do produto. O administrador deve encomendar o tecido e mandar diretamente para o time de produção.

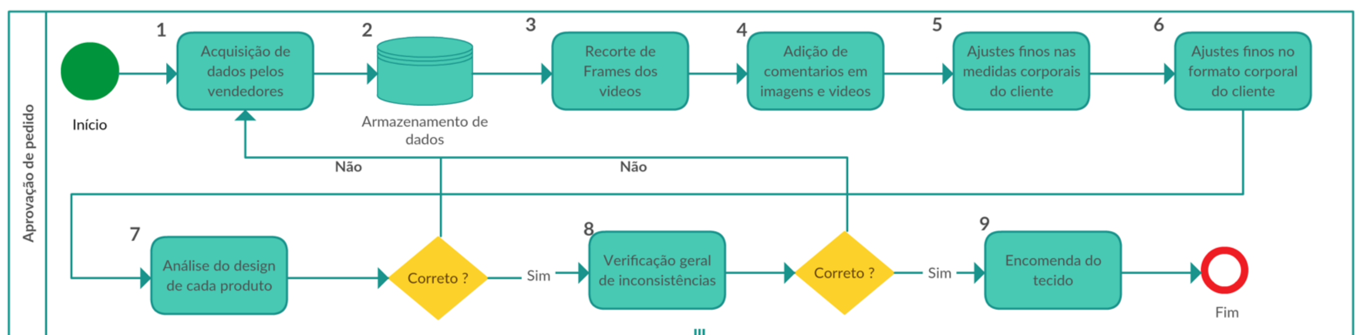

Figura 23: Fluxo de atividade (BPMN)

4.2 Visão detalhada do processo de negócio – Camada de Visão

O processo descrito deve ser realizado cuidadosamente para cada pedido e leva em média 30 (trinta) minutos para ser concluído, alocando bastante tempo dos administradores. Uma das causas dessa demora está relacionada com o desempenho do sistema, já que a arquitetura atual faz com que muitos dados sejam recarregados repetidamente a cada ação tomada pelo usuário.

Em outras palavras, para que se troque de página ou que se execute determinada ação dentro do sistema é necessário, muitas vezes, que se recarregue toda a interface: menu, cabeçalho, corpo da página, arquivos *javascript*, arquivos de estilo etc. Essa prática era muito comum no início dos sistemas *Web*, quando não se podia utilizar chamadas *AJAX* e atualizar páginas dinamicamente. O sistema utiliza chamadas *AJAX* em algumas funcionalidades, mas ainda não resolve a necessidade de que se recarregar a página como um todo durante a navegação entre diferentes páginas do sistema.

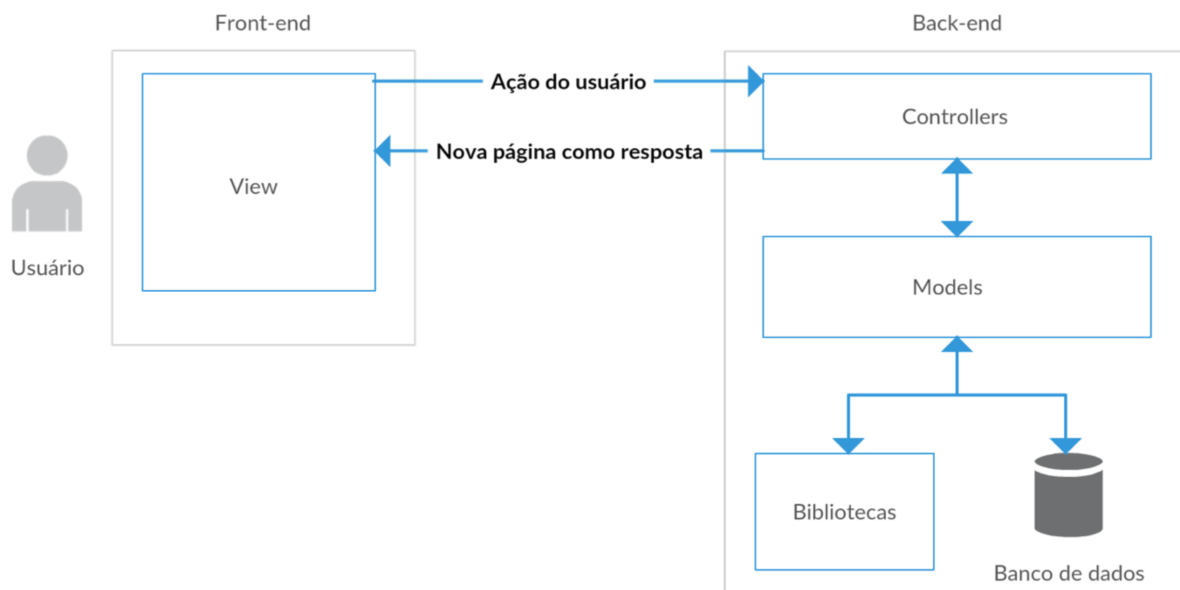

Figura 24: Fluxo de chamadas

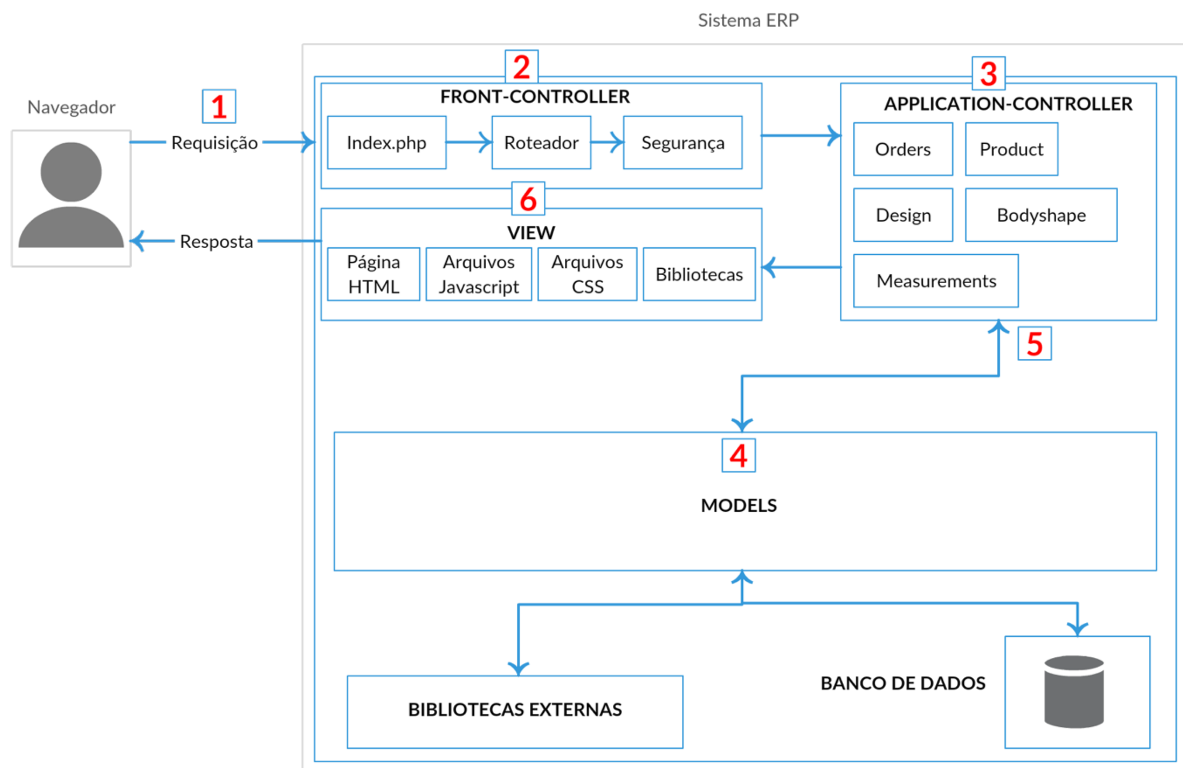

Figura 25: Fluxo detalhado de chamadas

4.3 Visão detalhada do processo de negócio – Camada de Negócio

As regras de negócio do sistema estão contidas na sua maioria na camada de modelo e dentro do banco de dados. Todas as requisições devem, obrigatoriamente, passar pelo mesmo fluxo interno (figura 25):

1. A requisição *HTTP* é lançada pela camada de visão do sistema (Passo 1 na Figura 25)
2. A camada mais externa do *Codeigniter (front-controller)* trata a requisição e através de meta-dados contidos nela a atribui ao controlador requisitado.
3. O controlador responsável deve então instanciar os modelos que possuem a lógica de negócio responsável por tratar a requisição e realizar as operações necessárias.
4. Dentro dos modelos existe código responsável pela lógica de negócio do sistema, desde a mais simples tarefa até as mais complicadas. Os modelos também podem utilizar bibliotecas externas ao framework para realizar sua lógica e também utilizam o banco de dados.
5. Após as operações dos modelos sejam concluídas, o fluxo de execução deverá voltar aos controladores, que por sua vez, irão utilizar os dados obtidos para gerar dinamicamente as visões do sistema.
6. Finalmente, os arquivos gerados (*HTML*, Javascript, CSS, etc.) serão enviados como resposta aos clientes e renderizados completamente na camada de visão.

4.4 Novo fluxo de execução

Com a nova arquitetura de SPA, o fluxo de execução do sistema será alterado. Com isso, a maior parte dessas mudanças será realizada na camada de visão, visto que é possível preservar grande parte dos controladores e modelos do sistema. O novo fluxo utilizará a capacidade de frameworks que implementam SPA de requisitar e renderizar somente o necessário durante o uso do sistema (figura 26).

Em outras palavras, a camada de visão se torna mais independente do servidor e ganhará a responsabilidade de construir as páginas do lado do cliente (antes construídas pelo servidor). A comunicação entre o cliente e o servidor será reduzida e também mais focada na ação tomada pelo usuário, comunicando-se basicamente por meio de chamadas *AJAX* para a captura e envio de dados. Esses dados deverão ser tratados pelo framework utilizado, e ele

então deverá gerar novas páginas dinamicamente, sem a necessidade de recarregar todo o sistema.

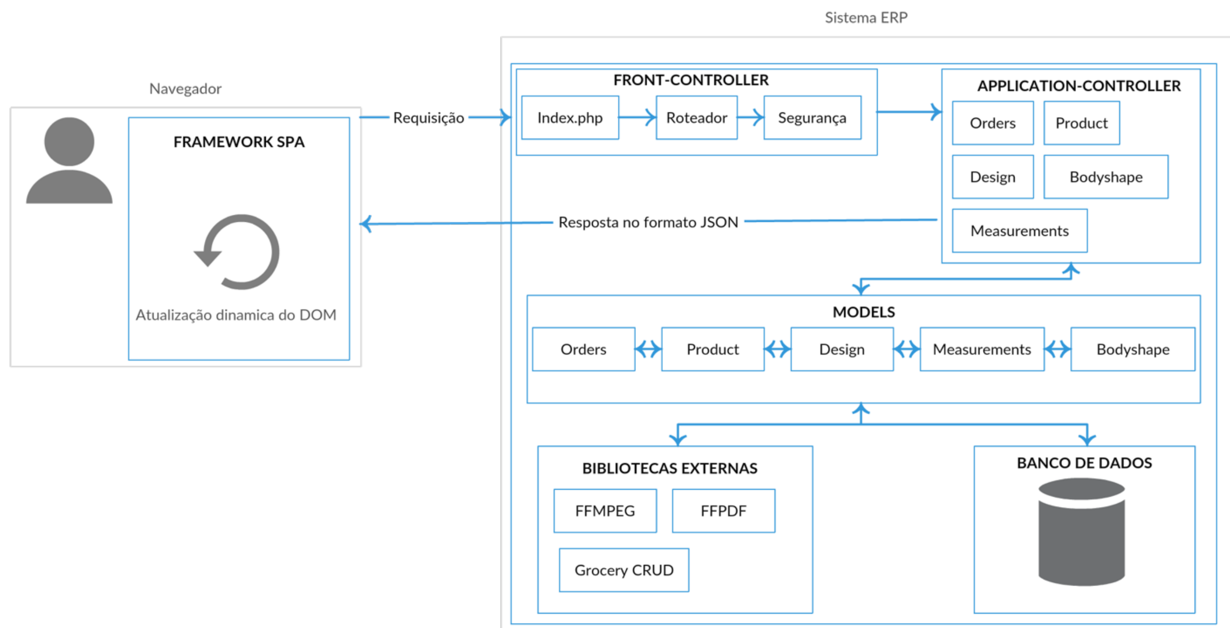

Figura 26: Fluxo de chamadas com framework SPA

4.5 Comparação de desempenho

Realizou-se uma comparação entre a arquitetura atual do sistema e a arquitetura proposta. A comparação foi feita utilizando o ambiente de desenvolvedor do navegador Firefox 57.0, no qual podemos calcular o tempo e o tamanho de cada requisição no sistema. O resultado consiste na média dos dados obtidos. Cada experimento foi realizado trinta e três vezes (33) e então obtida a média de cada métrica. O servidor utilizado para as execuções foi o servidor de testes da empresa, o qual possui a mesma capacidade de processamento, memória e rede do servidor de produção (localizado na cidade de Melbourne, Austrália). As chamadas para cada página foram efetuadas em uma rede de 50 Mbps, da cidade de João Pessoa (Brasil).

O uso de um framework *SPA* requer que grande parte ou todas as dependências sejam carregadas inicialmente no sistema (o que pode levar algum tempo). Após essa carga inicial das dependências, o cliente passará a somente requisitar informações específicas do que o usuário esta fazendo no sistema, evitando o envio de todas as dependências novamente a cada requisição e, consequentemente, melhorando o tempo de resposta e o tamanho delas (ver tabelas 1 e 2).

Tabela 1: Desempenho da arquitetura **a**tual

CODEIGNITER - atual		
Página	**Tempo total médio das requisições**	**Tamanho (Kb)**
Menu principal do pedido	7,18	346,10
Pagina de imagens	7,35	193,20
Ferramenta de recorte de frames	21,83	543,34
Aprovação de medidas	21,87	559,80

Tabela 2: Desempenho estimado da arquitetura proposta

Proposta		
Página	**Tempo total médio das requisições**	**Tamanho (Kb)**
Carregamento Inicial do sistema	34,40	1442,23
Menu principal do pedido	1,81	77,70
Pagina de imagens	1,19	4,30
Ferramenta de recorte de frames	1,52	5,40
Aprovação de medidas	1,74	152,30

A nova proposta possui um tempo médio estimado de navegação entre páginas menor do que a arquitetura atual, o que torna o sistema mais responsivo, rápido e confortável de se utilizar. Por outro lado, a nova arquitetura requer um carregamento inicial consideravelmente grande (tabela 2), uma vez que todas as bibliotecas e dependências do sistema são carregados de uma vez durante a inicialização do sistema. Apesar disso, esse carregamento faz com que todas as outras requisições ao servidor sejam bem mais leves, como observado na figura 27 e 28. A figura 28 mostra em detalhes o resultado de cada teste efetuado. Podemos observar que todas as requisições efetuadas com a técnica *SPA* superam as requisições efetuadas com a tecnologia tradicional em termos de velocidade.

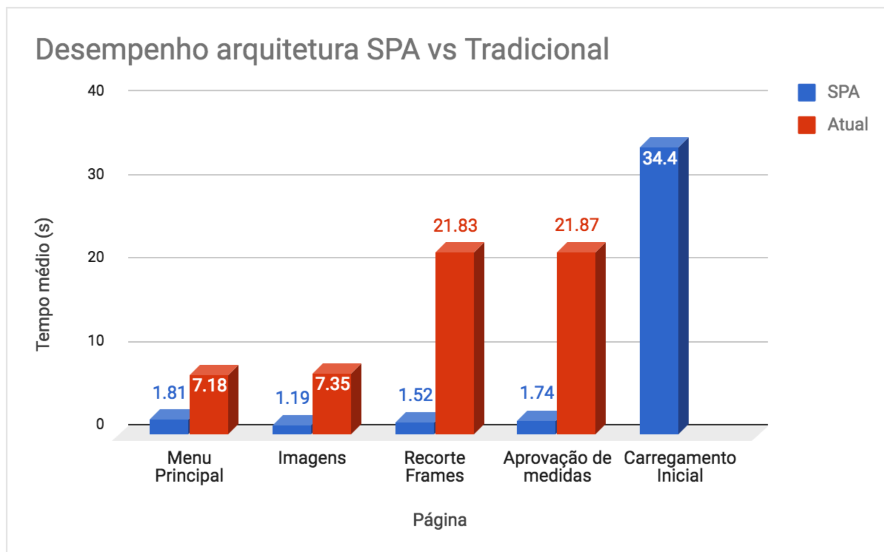

Figura 27: Comparação de desempenho médio das arquiteturas SPA e Tradicional

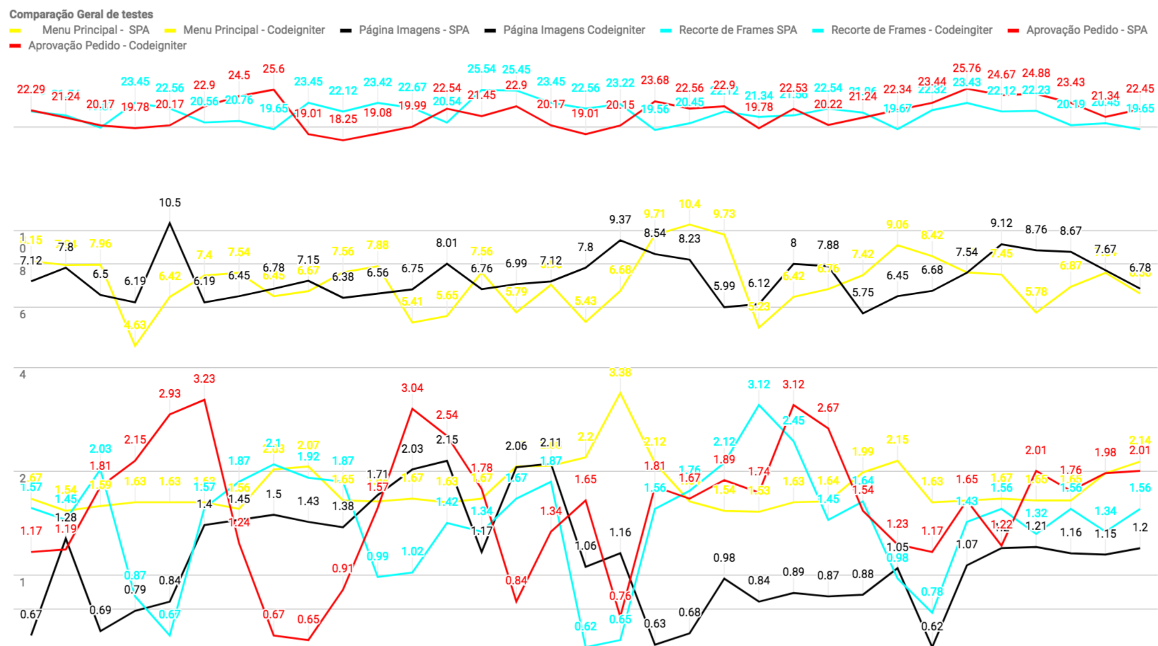

Figura 28: Comparação de desempenho geral das arquiteturas SPA e Tradicional

4.6 Análise de evolução para Micro-serviço

O sistema ERP em questão possui características de sistemas monolíticos [5]:

a) Acoplamento forte entre serviços: Uma vez que a comunicação desses serviços se dá por chamadas em um mesmo processo.
b) Distribuição dos serviços e banco de dados em um único servidor: Sistemas monolíticos são caracterizados por reter todos os elementos do sistema em um único *container*.
c) Baixa tolerância a falhas: Caso algum componente falhe, não existe um mecanismo eficiente para que o serviço se recupere e volte a funcionar automaticamente.
d) Incapacidade de crescimento horizontal: Uma vez que a arquitetura não permite um crescimento específico de cada serviço e nem a capacidade de replicação individual de cada serviço.

Essas características podem causar instabilidade, lentidão e ainda indisponibilidade do sistema como um todo caso haja necessidade de realizar manutenção em qualquer de suas partes [5]. Tais características atrapalham o funcionamento do sistema e, consequentemente a qualidade do serviço prestado pela empresa. Esse quadro poderia ser revertido com o uso de uma arquitetura de Micro-serviços, na qual, os processos da empresa tornam-se mais independentes, replicáveis e escaláveis [5]. Dessa forma, o sistema ganharia maior capacidade de processamento para serviços mais pesados (tratamento de imagens e vídeos por exemplo), maior disponibilidade de serviços críticos (submissão de pedidos) e criaria uma maior resiliência em caso de falhas, já que seria possível ter diversas instâncias de um único serviço [5].

Para isso, seria necessário, entre outros fatores, que os serviços alterassem a forma de comunicação entre eles, uma vez que não estariam mais presentes no mesmo processo. Essa comunicação poderia se dar por meio de chamadas *HTTP* ou outro mecanismo de invocação remota. Migrar toda a arquitetura do sistema é uma tarefa bastante custosa, dessa forma, essa transição deve ser realizada gradativamente, de forma que os serviços mais custosos em termos de uso de recursos e importantes da empresa passem por essa mudança primeiramente.

Podemos citar três estratégias para migrar uma arquitetura tradicional para uma baseada em Micro-serviços.

A primeira consiste em transformar todos serviços existentes em Micro-serviços, mudando quase que completamente a arquitetura. Essa arquitetura tornaria o sistema como um todo bastante resistente á falhas, dado que todos os serviços teriam completa independência entre si (figura 27). Além disso, seria possível escalar cada serviço á medida

que é necessário, evitando ter que escalar o sistema como um todo [5]. Essa opção pode custar bastante tempo para ser realizada, visto que praticamente todo o sistema teria que sofrer mudanças drásticas no seu projeto.

Com essa estratégia, poderíamos também utilizar serviços com tecnologias e até mesmo banco de dados diferentes. Por exemplo, o serviço de fornecimento de imagens poderia ser construído utilizando Java por apresentar um desempenho bastante elevado. Além disso, poderíamos usar outros tipos de banco de dados, por exemplo *NoSql,* para manipular os dados relacionados com os pedidos, facilitando a recuperação, inserção e modificação de dados.

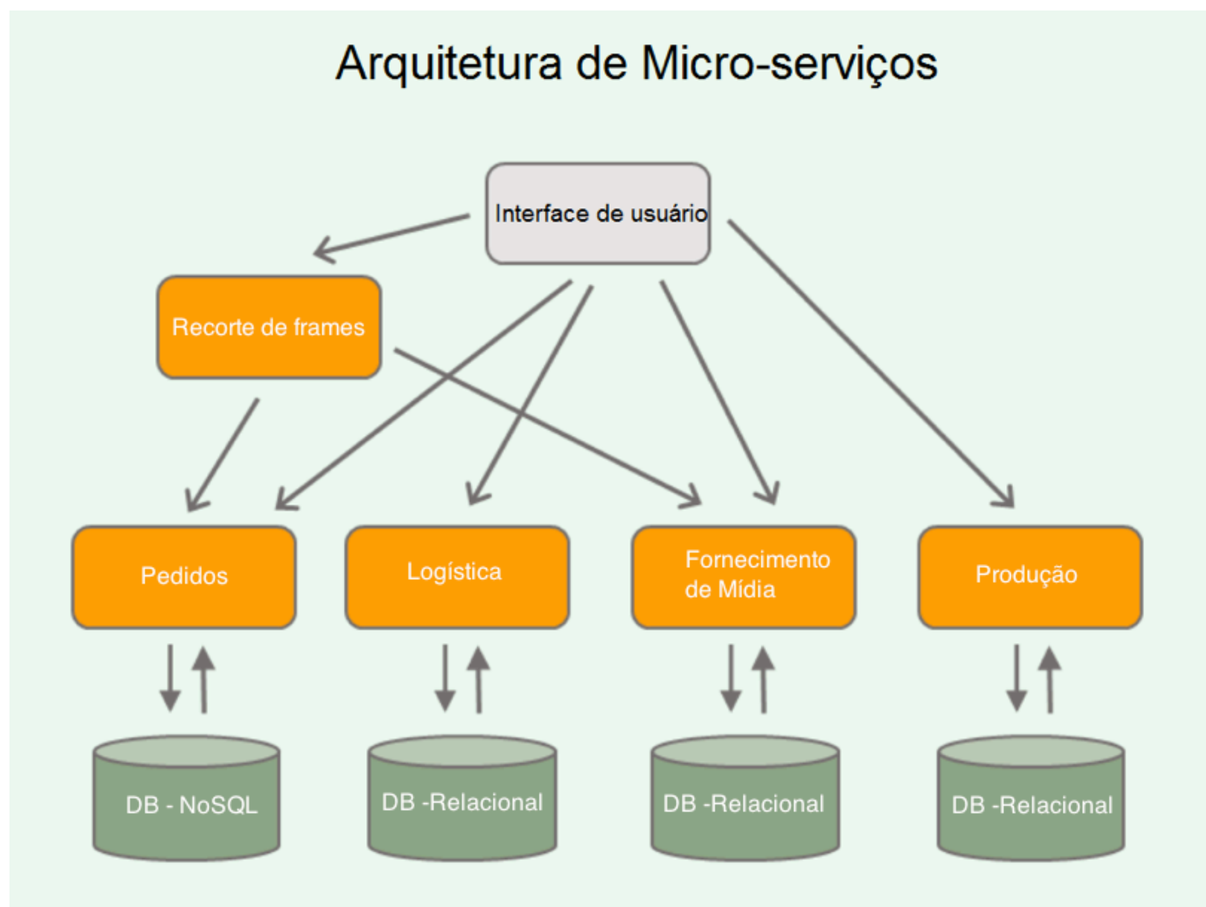

Figura 29: Arquitetura de Micro-Serviços
Fonte: https://dzone.com/articles/what-are-microservices-actually. Acesso em: 30 nov. 2017

A segunda estratégia consiste em refatorar os serviços mais custosos do sistema, melhorando a escalabilidade e disponibilidade. Isso poderia ser feito criando um *container* a parte para esses serviços. Dessa forma, funcionariam de maneira independente do restante do sistema (monolítico) [5]. Essa estratégia consiste em criar redirecionamentos das requisições no lado do servidor, assim, permitindo que o servidor principal tenha mais disponibilidade de

processamento para outras tarefas. Essa estratégia permite que os serviços mais antigos sejam mantidos no servidor monolítico e que novos serviços sejam adicionados. Dessa forma, serviços mais custosos, como o de fornecimento de imagens e vídeos pode ser isolado da aplicação monolítica, não interferindo no desempenho geral do sistema.

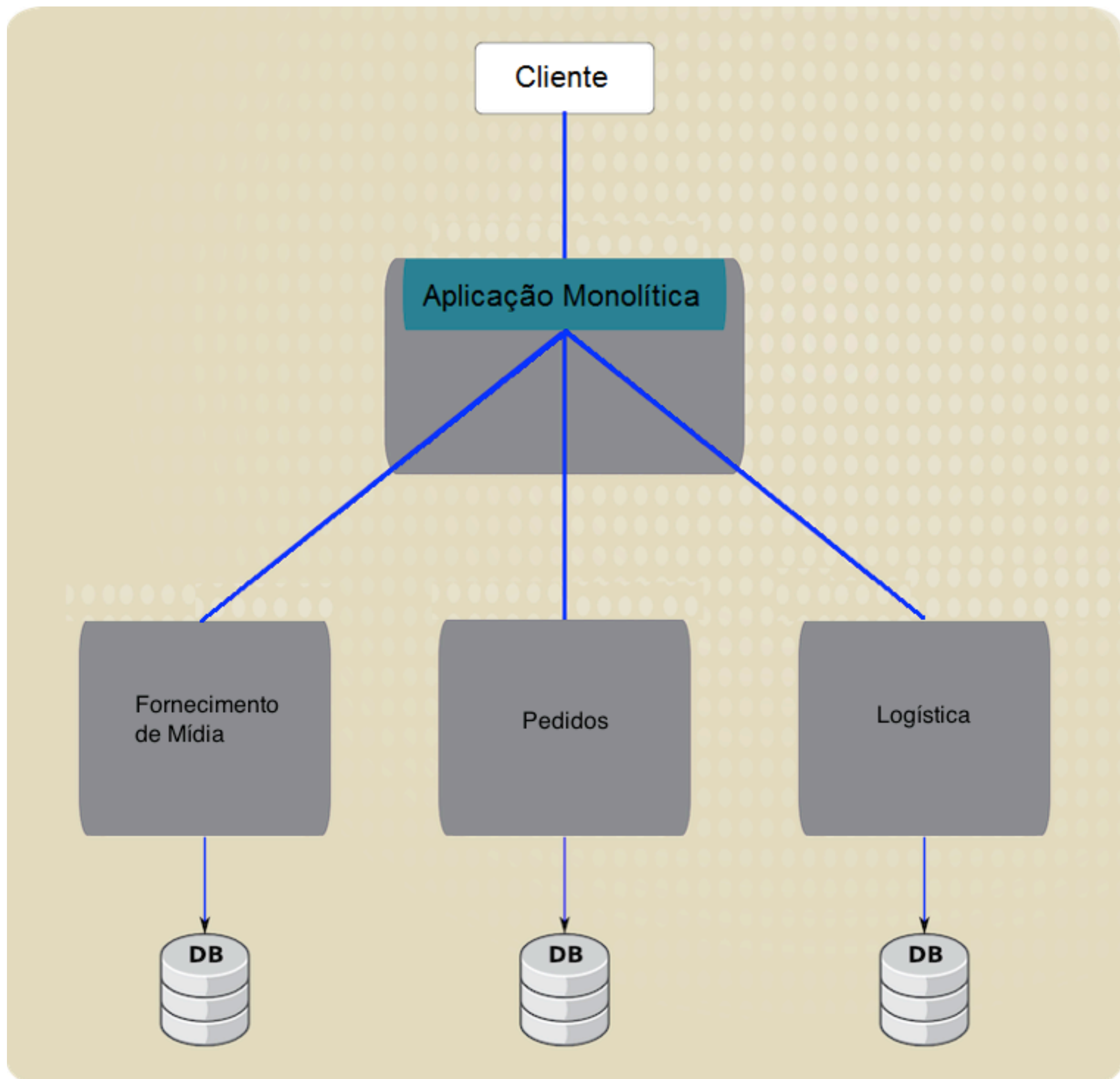

Figura 30: Arquitetura de Híbrida de Micro-Serviços
Fonte: https://access.redhat.com/documentation/en-us/reference_architectures/2017/html-single/microservice_architecture/index. Acesso em: 30 nov. 2017

A terceira estratégia é bastante similar a primeira, embora o redirecionamento é efetuado no lado do cliente. Assim, as requisições para os serviços seriam redirecionadas por um *gateway*, ver figura 29. Dessa forma, os serviços mais custosos do sistema ganhariam completa independência do servidor principal e também não seria necessário o uso de um

mecanismo de redirecionamento no lado do servidor. Alguns serviços, tais como os de Pedidos e de Logísticas poderiam funcionar independente da aplicação monolítica, garantindo maior disponibilidade para submissão de pedidos e também para os processos de logística interno e externos da empresa. Estas características, tornam o sistema mais tolerante á falhas, visto que o redirecionamento para diferentes servidores acontece na camada de visão.

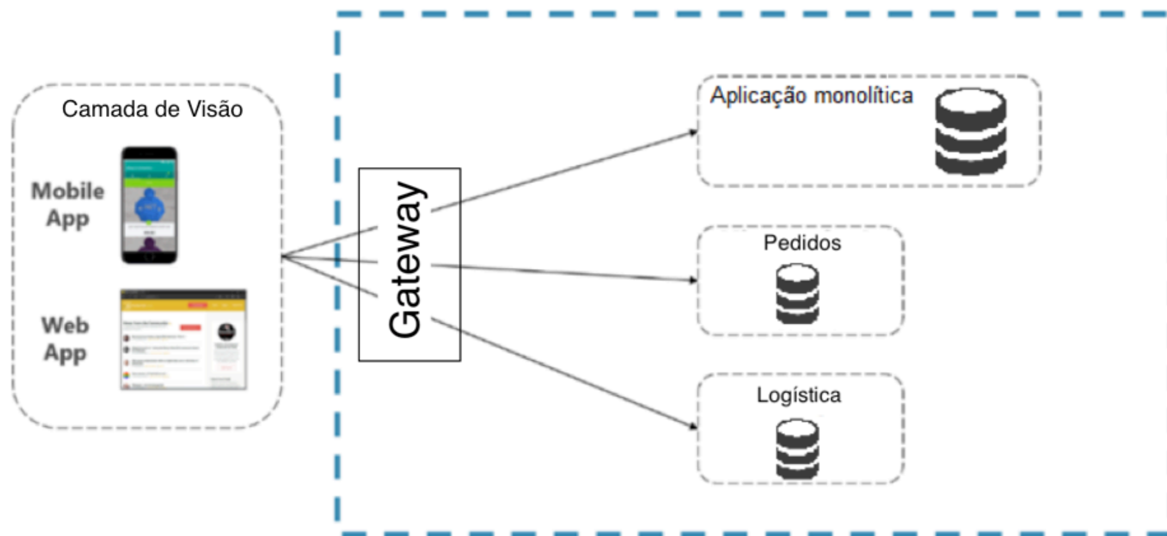

Figura 31: Arquitetura de Híbrida de Micro-Serviços com chamadas diretas pelos clientes

Fonte: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern. Acesso em: 30 nov. 2017

Uma arquitetura baseada em Micro-serviços trás inúmeros benefícios para o sistema, embora deva ser implantada com muita cautela, já que leva a uma refatoração de parte do código existente e também adiciona mais complexidade ao sistema como um todo [5]. É sugerido [5] que as mudanças sejam realizadas gradativamente, priorizando os serviços mais importantes. Entretanto, este trabalho se restringiu a propor melhorias futuras na arquitetura do sistema e, devido o tempo e escopo, não realizou implementação e avaliações dessas propostas.

5. Considerações Finais e Trabalhos Futuros

Sistemas de informação tornam-se cada vez mais essenciais para o funcionamento ágil e correto de uma empresa. Sem eles seria muito difícil manter bons padrões de qualidade e velocidade nos processos de negócio.

Este trabalho realizou uma análise das principais tecnologias e arquiteturas utilizadas no desenvolvimento *Web* – desde os mais primitivos mecanismos utilizados no surgimento da Internet até as mais avançadas técnicas de desenvolvimento com Micro-serviços. Além disso, foi feita uma análise do sistema ERP em questão por meio de engenharia de requisitos e projeto de software.

Durante essa pesquisa foi constatado que a arquitetura atual do sistema pode evoluir em diversos aspectos, desde o funcionamento da camada de visão do sistema, até possíveis mudanças nas camadas presentes no lado do servidor. A comparação de desempenho efetuada entre o modelo de arquitetura tradicional e o modelo de *SPA* constatou que existe um ganho substancial de desempenho quando utilizada a arquitetura *SPA* na camada de visão. Tal ganho permite que a empresa consiga suportar uma maior quantidade de clientes, permitindo uma maior expansão dos seus serviços, e sugere também, uma melhor experiência do usuário.

Como trabalhos futuros, é sugerido a implementação e avaliação dos cenários propostos na análise de evolução da arquitetura do sistema. A ideia seria evoluir a atual arquitetura dos modelos e controladores, que hoje possuem características de um sistema monolítico para uma arquitetura mais desacoplada, como um sistema *SOA,* mais especificamente, com algumas características de uma arquitetura de Micro-serviços, tais como, decomposição de módulos, encapsulamento de serviços, contextos independentes para cada serviço, abstração de entidades e processos, entre outras. Uma definição mais detalhada de quais cenários seriam implementados e um planejamento das atividades de migração também deveria ser considerado.

Referências

[1] FIELDS, DUANE KOLB, MARK A. Web development with JSP. Greenwich, Manning, 2002.
[2] HUNTER, JASONCRAWFORD, WILLIAM. Java Servlet Programming. 2. ed. 2001.

[3] HOLDENER T. Ajax The definitive guide. 1. ed. 2008.

[4] SOMMERVILLE, IAN Software Engineering. 9. ed. 2011

[5] NEWMAN, S. Building microservices. 1. Ed. 2015

[6] ERL, T. SOA PRINCIPLES OF SERVICE DESIGN. 1. Ed. 2007

[7] CLARK, KIM.Microservices, SOA, and APIs: Friends or enemies?, 2016. Disponível em:<https://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html/>. Acesso em: 31 de jul. 2017

[8] HAINES S, DEVS ARE FROM VENUS, OPS ARE FROM MARS, PT. 3 – SOAP AND SERVICE-ORIENTED ARCHITECTURE, 2014. Disponível em <https://turbonomic.com/blog/on-technology/devs-venus-ops-mars-pt-3-soap-service-oriented-architecture/>. Acesso em 1 de ago. 2017

[9] WEBBER, J. REST in Practice: Hypermedia and Systems Architecture. 1. Ed. 2010

[10] LEWIS, JAMESFOWLER, MARTIN. Microservices. martinfowler.com. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 5 ago. 2017.

[11] FOWLER, MARTIN. GUI Architectures. martinfowler.com, 2006. Disponível em: <https://www.martinfowler.com/eaaDev/uiArchs.html>. Acesso em: 7 ago. 2017.

[12] WALTHER, S. The Evolution of MVC. Stephenwalther.com. 2008. Disponível em: <http://stephenwalther.com/archive/2008/08/24/the-evolution-of-mvc>. Acesso em: 7 ago. 2017.

[13] GAROFALO, R. Building enterprise applications with Windows Presentation Foundation and the model view viewmodel pattern. Sebastopol, Calif.: O'Reilly Media, Inc., 2011.

[14] JAGGAVARAPU, MANOJ. Presentation Patterns : MVC, MVP, PM, MVVM. 2012, Disponível em: <https://manojjaggavarapu.wordpress.com/2012/05/02/presentation-patterns-mvc-mvp-pm-mvvm/>. Acesso em: 8 ago. 2017.

[15] FOWLER, MARTIN. Presentation Model. martinfowler.com. 2004, Disponível em: <https://martinfowler.com/eaaDev/PresentationModel.html>. Acesso em: 9 ago. 2017.

[16] SHAW, M.CLEMENTS, P. The golden age of software architecture. IEEE Software, n. 2, 2006.

[17] SLATER, NATE. Using Containers to Build a Microservices Architecture. Medium. Disponível em: <https://medium.com/aws-activate-startup-blog/using-containers-to-build-a-microservices-architecture-6e1b8bacb7d1>. Acesso em: 18 ago. 2017.

[18] SANTOS, RICARDO. Web Services and Micro services - Polarising. Polarising. 2015 Disponível em: <https://www.polarising.com/web-services-micro-services/>. Acesso em: 18 ago. 2017.

[19] FORD Neal. Art of Java Web development, chapter State-of-the-art web design. Manning, 2004.

[20] SIPOS, DANIEL. CodeIgniter: Getting Started. 2013. Disponível em: <https://www.digitalocean.com/community/tutorials/codeigniter-getting-started-with-a-simple-example>. Acesso em: 20 ago. 2017.

[21] CodeIgniter Web Framework. Codeigniter.com. 2014 Disponível em: <https://codeigniter.com/>. Acesso em: 22 ago. 2017.

[22] MUNRO, JAMIE. Knockout.js. 1. ed. 2014.

[23] TATROE, KEVIN, MACINTYRE, PETERLERDORF, RASMUS. Programming PHP. Sebastopol, CA: O'Reilly Media, 2014.

[24] LOCKHART, JOSH. Modern PHP. Sebastopol: O'Reilly & Associates, 2015.

[25] Usage Statistics and Market Share of PHP for Websites, September 2017. W3techs.com. Disponível em: <https://w3techs.com/technologies/details/pl-php/all/all>. Acesso em: 7 set. 2017.

[26] SILBERSCHATZ, ABRAHAM, KORTH, HENRY FSUDARSHAN, S. Database systems concepts. Estados Unidos: McGraw-Hill Companies, Inc., 2011.

[27] AVEDA, SCOTT. What is the importance of a database management system. www.linkedin.com. Disponível em: <https://www.linkedin.com/pulse/what-importance-database-management-system-scott-aveda/>. Acesso em: 9 set. 2017.

[28] SULLIVAN, DAN. NoSQL for mere mortals. Boston: Addison-Wesley, 2015.

[29] FOWLER, MARTIN. Introduction to NoSQL. In: [s.l.: s.n.], 2012.

[30] FOWLER, MARTIN. Richardson Maturity Model. martinfowler.com. Disponível em: <https://martinfowler.com/articles/richardsonMaturityModel.html>. Acesso em: 10 set. 2017.

[31] DE SOUZA, ARICÉLIOCÂNDIDO, PETRÔNIO. Comparativo técnico de tecnologias de banco de dados: Relacional, NoSQL e NewSQL. Disponível em: <https://www.academia.edu/6559619/Comparativo_T%C3%A9cnico_entre_tecnologias_de_banco_de_dados_relacional_NoSQL_e_NewSQL>. Acesso em: 11 set. 2017.

[32] BREWER, E. Towards Robust Distributed. 2000. Disponível em: <http://www.cs.berkeley.edu/ brewer/cs262b-2004/PODC-keynote.pdf>.

[33] ANTONIO, GUTIERRY. Conheca a geracao de banco de dados NoSQL e NewSQL. DevMedia. Disponível em: <http://www.devmedia.com.br/conheca-a-geracao-de-banco-de-dados-nosql-e-newsql/33202>. Acesso em: 11 set. 2017.

[34] SIMONS, PETER. FastCGI — The Forgotten Treasure. Nongnu.org. Disponível em: <http://www.nongnu.org/fastcgi/>. Acesso em: 12 set. 2017.

[35] Best PHP Frameworks In 2017 – Level Up! – Medium. Medium. Disponível em: <https://medium.com/level-up-web/best-php-frameworks-for-web-developers-in-2017-c8a041671a79>. Acesso em: 12 set. 2017.

[36] PEDRO, BRUNO. Is REST better than SOAP? Yes, in some use-cases. Nordic APIs. Disponível em: <https://nordicapis.com/rest-better-than-soap-yes-use-cases/>. Acesso em: 15 set. 2017.

[37] What is Object/Relational Mapping? - Hibernate ORM. Hibernate.org. Disponível em: <http://hibernate.org/orm/what-is-an-orm/>. Acesso em: 22 out. 2017.

[38] JQUERY.ORG, JQUERY. jQuery. Jquery.com. Disponível em: <https://jquery.com/>. Acesso em: 27 out. 2017.

[39] JQUERY.ORG, JQUERY. jQuery UI. Jqueryui.com. Disponível em: <https://jqueryui.com/>. Acesso em: 27 out. 2017.

[40] OTTO, MARK. Bootstrap. Getbootstrap.com. Disponível em: <http://getbootstrap.com/>. Acesso em: 27 out. 2017.

[41] Taggd - Add notes to your images, responsively. Timseverien.com. Disponível em: <https://timseverien.com/taggd/v3/>. Acesso em: 27 out. 2017.

[42] Introduction to Raphaël.js - HTML5 Rocks. HTML5 Rocks - A resource for open web HTML5 developers. Disponível em: <https://www.html5rocks.com/en/tutorials/raphael/intro/>. Acesso em: 27 out. 2017.

[43] FPDF. Fpdf.org. Disponível em: <http://www.fpdf.org/>. Acesso em: 27 out. 2017.

[44] FFmpeg. Ffmpeg.org. Disponível em: <https://www.ffmpeg.org/>. Acesso em: 27 out. 2017.

[45] Grocerycrud.com. Disponível em: <https://www.grocerycrud.com/>. Acesso em: 27 out. 2017.

[46] MIKOWSKI, Michael S. and POWELL, Josh C. Single page web applications: JavaScript end-to-end. Shelter Island, NY: Manning, 2014.

[47] Single-page Applications | Code School. Code School. Disponível em: <https://www.codeschool.com/beginners-guide-to-web-development/single-page-applications>. Acesso em: 8 nov. 2017.

[48] PAREKH, ARPIT. Introduction on Data Warehouse with OLTP and OLAP. International Journal Of Engineering And Computer Science, v. 2, n. 8, p. 2569-2573, 2013.

Glossário

Termo	Definição
Design do produto	Atributos da roupa escolhidos pelo cliente em conjunto com os vendedores.
Erros de produção	Erros gerados durante ou logo antes do processo de manufatura da roupa.
Forma Corporal de produção	Painel físico, localizado na fábrica; A roupa é feita com base nesse painel. Ele norteia, juntamente com medidas e o padrão corporal dos clientes, como a roupa deve ser produzida.
Medidas dos clientes	Medidas corporais dos clientes, adquiridas durante o processo inicial de venda.
Padrão Corporal dos clientes	Características do corpo de cada cliente, adquiridas durante o processo inicial de venda.
PDF de produção	Artefato utilizado pela fábrica para produzir os produtos. Ele geralmente é utilizado em formato de papel e contém os principais dados necessários para produzir o produto.
Processo de correção	Processo feito após a confecção do produto. Esse processo visa corrigir ou adicionar detalhes finos aos produtos finais (Erros ou ajustes finos são identificados no processo de prova).
Processo de provas	Processo de análise do produto no corpo do cliente. Esse processo visa identificar possíveis erros ou ajustes a serem feitos (Ajustes feitos no processo de correção).
Usuário Administrador	Usuário responsável por efetuar trabalhos analíticos dentro da empresa. Tais como: Identificar possíveis erros antes da produção, adicionar mais detalhes de produção, organizar logística, etc.
Usuário Alfaiate	Responsável por efetuar o processo de correção de roupas e produzir relatórios com informações úteis de produção.
Usuário de produção	Responsáveis por todo o processo de produção na fábrica e também coordena processos de tradução de dados para língua nativa da fabrica e correção de roupas.

Usuário Vendedor	Responsável pelo processo inicial de venda e também por processos de prova. Ele também pode captar dados de pagamento dos clientes.