UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ ENGENHARIA DE SOFTWARE

THIAGO PEREIRA ROSA

UM MÉTODO PARA O DESENVOLVIMENTO DE SOFTWARE BASEADO EM MICROSSERVIÇOS

QUIXADÁ - CE
2016

THIAGO PEREIRA ROSA

UM MÉTODO PARA O DESENVOLVIMENTO DE SOFTWARE BASEADO EM MICROSSERVIÇOS

Trabalho de conclusão de curso submetido à Coordenação do Curso Bacharelado em Engenharia de *Software* da Universidade Federal do Ceará como requisito parcial para obtenção do grau de bacharel.
Área de concentração: Ciência da Computação/ Metodologia e Técnicas da Computação / Engenharia de *Software*.

Orientador(a): Ticiana Linhares Coelho da Silva.
Co-Orientador: Flávio Rubens de Carvalho Souza.

QUIXADÁ - CE

2016

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca do Campus de Quixadá

R696m Rosa, Thiago Pereira
Um método para o desenvolvimento de software baseado em microsserviços/ Thiago Pereira Rosa. – 2016.
64 f.: il. color., enc.; 30 cm.

Monografia (graduação) – Universidade Federal do Ceará, Campus Quixadá, Curso de Engenharia de Software, Quixadá, 2016.
Orientação: Profª. Me. Ticiana Linhares Coelho da Silva
Coorientação: Prof. Dr. Flávio Rubens de Carvalho Sousa
Área de concentração: Computação

1. Software - desenvolvimento. 2. Arquitetura de software. 3. Sistemas de computação. I. Universidade Federal do Ceará (Campus Quixadá). II. Título.

CDD 005.1

THIAGO PEREIRA ROSA

UM MÉTODO PARA O DESENVOLVIMENTO DE *SOFTWARE* BASEADO EM MICROSSERVIÇOS

Trabalho de conclusão de curso submetido à Coordenação do Curso Bacharelado em Engenharia de *Software* da Universidade Federal do Ceará como requisito parcial para obtenção do grau de bacharel.
Área de concentração: Ciência da Computação / Metodologia e Técnicas da Computação / Engenharia de *Software.*

Aprovado em: 15 / fevereiro / 2016

BANCA EXAMINADORA

Prof(a). MSc. Ticiana Linhares Coelho da Silva

Orientador(a)

Prof. Dr. Flávio Rubens de Carvalho Sousa

Co-Orientador

Prof. MSc. Régis Pires Magalhães

Universidade Federal do Ceará

Dedico este trabalho à minha família e amigos.

RESUMO

Devido ao grande avanço da tecnologia, as formas de comunicação e conectividade se reinventaram. Assim, a maneira de desenvolver *software* tem se tornado mais complexa e multidisciplinar. Utilizar a arquitetura monolítica como padrão de desenvolvimento de aplicações corporativas é uma abordagem comumente empregada, no entanto, este padrão gera complicações quando utilizado para desenvolver grandes aplicações, com contratempos na escala e principalmente inconvenientes quando há necessidade de iterações e entregas frequentes. Com isso, a arquitetura de microsserviços surge como uma alternativa para construção de aplicações corporativas complexas. Fundamentado nesta alternativa, este trabalho de conclusão de curso propõe e avalia um método para o desenvolvimento de *software* baseado em microsserviços. Para isso, foi realizado um estudo de caso construindo um sistema baseado em microsserviços para complementar o tratamento medicinal com aplicação de fototerapia.

Palavras chave: *Software* como Serviço, Desenvolvimento de *Software*, Sistema Monolítico, Arquitetura orientada a Serviços.

ABSTRACT

Due to the great advancement of technology, communication tools and connectivity have reinvented. Therefore, the way to develop software has become more complex and multidisciplinary. Using the monolithic architecture as standard enterprise application development is an approach commonly employed, however, this pattern generates complications when used to develop large applications with setbacks in scale and particularly inconvenient when it is necessary iterations and frequent deliveries. Indeed, the microservices architecture is an alternative for building complex enterprise applications. Based on this alternative, this term paper proposes and evaluates a method for developing a microservices software. For this purpose, was conducted a case study building a microservices-based system to complement the medical treatment with phototherapy application.

Keywords: Software as a Service, Software Development, Monolithic Application, Service-Oriented Architecture.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A evolução da tecnologia da informação (TI) e a melhoria da conectividade da internet permitiram uma revolução na forma como as aplicações e os serviços de TI são construídos e disponibilizados aos usuários. Nesse contexto, a computação em nuvem dispõe, a um custo acessível, de uma infraestrutura escalável, com qualidade de serviço e altamente disponível. Desse modo, a forma de desenvolver *software* também tem evoluído para melhorar a construção de *software* complexo e confiável de maneira modular.

Uma das maneiras de se construir um *software* a partir do contexto de negócios é utilizar o *Domain-Driven Design* ou *design* orientado ao domínio. É necessário entender os problemas de negócio, bem como o funcionamento das aplicações corporativas e planejar um cenário que se assemelhe ao mesmo, para que, em seguida, seja possível desenvolver um produto de *software* que atenda a estes requisitos.

Atualmente, há diversos padrões comumente difundidos para o desenvolvimento de aplicações corporativas, visto que as fases do desenvolvimento não são atividades triviais. A arquitetura monolítica é um padrão amplamente usado para o desenvolvimento de aplicações corporativas. Esse padrão funciona razoavelmente bem para pequenas aplicações, pois o desenvolvimento, testes e implantação de pequenas aplicações monolíticas são relativamente simples (FOWLER; LEWIS, 2014). No entanto, para aplicações grandes, complexas e muito acopladas, a arquitetura monolítica torna-se um obstáculo ao desenvolvimento e implantação, além de dificultar a entrega contínua e limitar a adoção de novas tecnologias. Para aplicações complexas, que necessitem de alta performance computacional e disponibilidade ininterrupta é mais interessante utilizar uma arquitetura de microsserviços, que divide a aplicação em um conjunto de serviços leves e coesos.

O termo *Microservices* ou Microsserviços surgiu nos últimos anos para descrever uma maneira particular de conceber aplicações de *software* como uma suíte de serviços independentemente implementáveis (FOWLER; LEWIS, 2014). Embora ainda não exista uma definição precisa deste estilo arquitetônico, pode-se elencar um conjunto de características comuns, tais como controle descentralizado de linguagens e de dados e independência dos serviços.

A crescente demanda por recursos descentralizados e orientados a dados em aplicações de *software* transformou esse tipo de aplicação, em muitos casos, em conglomerados complexos de serviços que operam diretamente em dados, e, em outros, em uma arquitetura gerenciável coerente.

O objetivo prevalecente de implementar um método de desenvolvimento baseado em microsserviços é minimizar a necessidade de re-implementação, caso ocorram mudanças, por meio da limitação de serviços devidamente coesos com os mecanismos projetados de evolução das interfaces de serviço, envolvendo uma combinação de tecnologias em diferentes áreas da tecnologia da informação, além de suportar o quesito multiplataforma, propendendo à interoperabilidade.

Nesse contexto, este trabalho propõe um método para o desenvolvimento de *software* baseado em microsserviços. Este método fornece um conjunto de passos para a construção de *software* utilizando microsserviços. Para avaliar este método foi realizado um estudo de caso envolvendo o desenvolvimento de um *software* baseado em microsserviços para suplementar o tratamento medicinal com aplicação de fototerapia.

Nesta proposta de pesquisa, o objetivo geral é propor e validar um método para o desenvolvimento de *software* baseado em microsserviços. A partir deste objetivo mais geral, foram elencados como objetivos específicos: (i) identificar modelos de desenvolvimento existentes baseados em microsserviços; (ii) identificar características e funcionalidades fundamentais em sistemas baseados em microsserviços; (iii) definir um método para o desenvolvimento de *software* baseado em microsserviços; e (iv) realizar um estudo de caso utilizando o método proposto.

No próximo capítulo é apresentada a fundamentação teórica relacionada e como as abordagens influem, em caráter prático, neste trabalho de conclusão de curso.

2 FUNDAMENTAÇÃO TEÓRICA

A seguir são apresentados os conceitos-chave adotados neste trabalho. Nas três primeiras subseções são explanados os conceitos de microsserviços e, posteriormente, seus princípios, os modelos arquiteturais tradicionais e baseados em microsserviços, a utilização do REST como modelo arquitetural, a componentização dos microsserviços, o *design* orientado a domínio e computação em nuvem, respectivamente. Na subseção seguinte é introduzido o conceito de arquitetura de *software*, aplicações monolíticas e resiliência, bem como são tecidas algumas considerações sobre o processo de desenvolvimento de *software* baseado em microsserviços.

2.1 Microsserviços

Newman (2015) explica sobre serviços autônomos que trabalham juntos e que, nos últimos dez anos, os sistemas distribuídos se tornaram mais refinados, evoluindo de aplicações monolíticas de código pesado para microsserviços independentes.

O termo microsserviços, ou *microservices*, é definido por Thönes (2015) como um aplicativo que pode ser implantado, dimensionado e testado de maneira independente, seguindo o princípio da responsabilidade única. O intuito da responsabilidade única é que o aplicativo desenvolvido deve ser planejado para realizar apenas um conjunto mínimo viável de tarefas, sendo facilmente compreendido, modificável e substituível.

O princípio da responsabilidade única, ou SOLID, baseia-se em requisitos funcionais, não funcionais, ou, como o autor aborda, em requisitos multifuncionais. Um exemplo de funcionamento de um microsserviço pode ser um processador estar lendo mensagens de uma fila na memória enquanto realiza uma pequena parte da lógica de negócios, podendo variar de algo funcional a não funcional com a responsabilidade única de servir a um determinado recurso em particular.

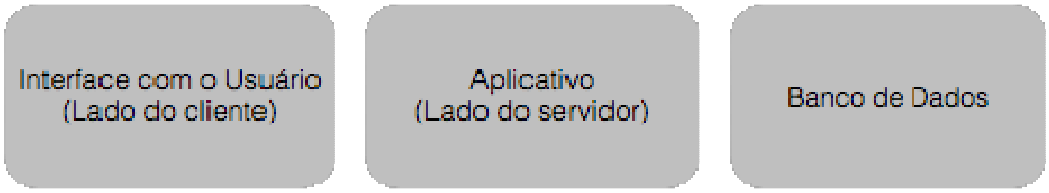
Os microsserviços devem ser focados, pequenos e executarem uma única tarefa por conta própria, decompondo funcionalmente a aplicação em um conjunto colaborativo de serviços, em que cada serviço implementa um conjunto de funções relacionadas de maneira bem restrita, para implementar as regras de negócio. O estilo arquitetônico baseado em microsserviços é uma abordagem para o desenvolvimento de uma aplicação única, baseada em um conjunto de microsserviços, cada um executando em seu próprio processo e se comunicando

por meio de um mecanismo leve, muitas vezes, uma API de recursos HTTP (FOWLER; LEWIS, 2014).

Os serviços são construídos em torno das regras de negócio da organização e são independentemente implementáveis por máquinas totalmente automatizadas. Contudo, o gerenciamento pode ser centralizado e escrito por meio de linguagens de programação diferentes e com diferentes mecanismos de persistência de dados.

Observando-se a Figura 1, a Interface do lado do cliente consiste em páginas desenvolvidas com tecnologia HTML, Javascript e CSS, em execução através de um navegador na máquina do usuário. A aplicação do lado servidor lida com as solicitações HTTP, executa a lógica de negócio do domínio, recupera e atualiza os dados na base de dados, seleciona e disponibiliza as visões HTML enviadas ao *browser* e geralmente, possui um SGBD relacional.

Figura 1 - Estilo arquitetônico básico baseado em microsserviços



Fonte: Elaborado pelo autor.

Para Fowler e Lewis (2014), o termo microsserviços surgiu nos últimos anos para descrever uma maneira peculiar de desenvolver aplicações de *software* independentemente implementáveis, como uma suíte de serviços, embora este estilo de desenvolvimento ainda não seja precisamente definido. Com os serviços independentemente implementáveis e escaláveis, é possível desenvolvê-los em diferentes linguagens de programação e também mantê-los e gerenciá-los por equipes distintas.

Outro fator abordado por Fowler e Lewis é a componentização via serviços, que trata os componentes como uma unidade de *software* autônoma, substituível e atualizável. Uma aplicação baseada em microsserviços pode utilizar bibliotecas, mas o intuito principal é modularizar o *software* em serviços independentes. Para os autores, bibliotecas são componentes que estão ligados a um determinado programa e que são chamados utilizando-se funções, enquanto os serviços estão completamente fora do processo de componentização e se comunicam através de um mecanismo de solicitação de serviço ou chamada a um procedimento remoto.

O objetivo prevalecente de implementar um método de desenvolvimento baseado em microsserviços é minimizar a necessidade de re-implementação, caso ocorram mudanças,

por meio da limitação de serviços devidamente coesos com os mecanismos projetados para evolução das interfaces do serviço.

2.1.1 Princípios dos Microsserviços

Newman (2015) elenca os princípios e objetivos estratégicos junto às práticas de *design* e *delivery* para o desenvolvimento de *software* baseado em microsserviços, defendendo a construção de serviços autônomos pequenos que trabalham em conjunto. Os princípios dos microsserviços são:

- A modelagem deve estar focada no domínio de negócio.
- Utilizar a cultura da automação.
- Abstrair os detalhes da implementação.
- Descentralizar todas as coisas.
- Isolar as falhas.
- *Deploy* independente.

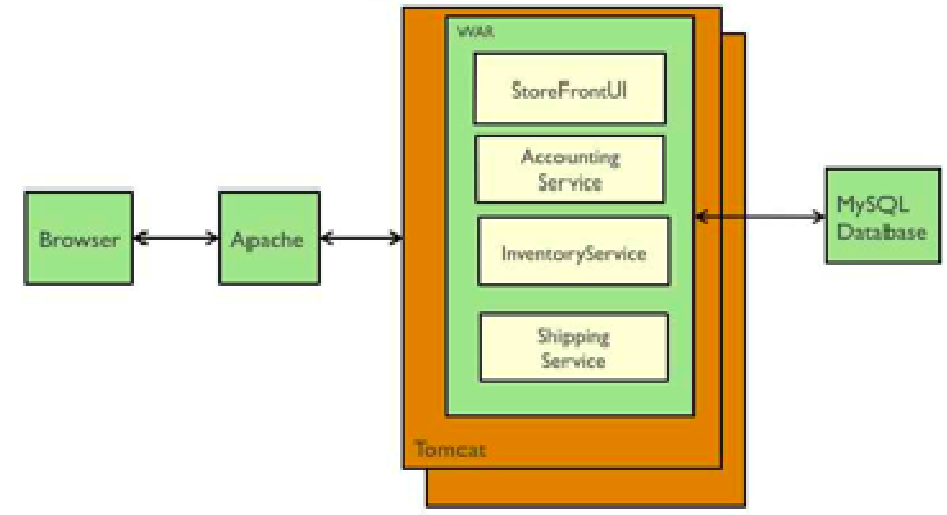
Os princípios abordados acima visam, segundo Newman (2015), reduzir a inércia, fazendo escolhas que favoreçam o *feedback* e mudanças rápidas, reduzindo as dependências entre as equipes. Para eliminar a complexidade acidental deve-se substituir os processos complexos, desnecessários e principalmente as integrações também demasiadamente complexas com o propósito de privilegiar o desenvolvimento.

2.1.2 Modelo de arquitetura web tradicional

À primeira vista, o contexto de uma aplicação *web* tradicional possui algumas vantagens em relação ao contexto baseado em microsserviços, como por exemplo nas fases de desenvolvimento e testes, em que as atuais ferramentas de desenvolvimento, comumente tituladas no setor de TI como IDE's ou *Integrated Development Environment*, oferecem apoio tanto ao desenvolvimento quanto ao trabalho em equipe do grupo de desenvolvedores. A facilidade de instalação, no caso de uma aplicação utilizando a tecnologia Java, aparentemente também é mais atrativa, visto que é necessária apenas a implantação de um arquivo WAR no contêiner *web*, por exemplo. Com isso, obtém-se simplicidade de escala seguindo o modelo citado, tornando possível replicar a aplicação executando várias cópias do aplicativo,

orquestradas por um balanceador de carga responsável por receber e direcionar as requisições oriundas dos clientes.

Figura 2 - Arquitetura tradicional de uma aplicação *web*



Fonte: Microservices architecture (2015).

Na Figura 2, é apresentada uma arquitetura tradicional de uma aplicação monolítica *web*, desenvolvida utilizando a tecnologia Java. A aplicação consiste em um único arquivo *WAR*, que pode ser implantado em um contêiner *web*, como neste exemplo do *Apache Tomcat*[1], um servidor de código fonte aberto responsável pelo processamento das tecnologias *Java Servlet* e *JavaServer Pages*, que pode ser executado por diversas instâncias conectadas a um balanceador de carga, a fim de ampliar a disponibilidade da aplicação.

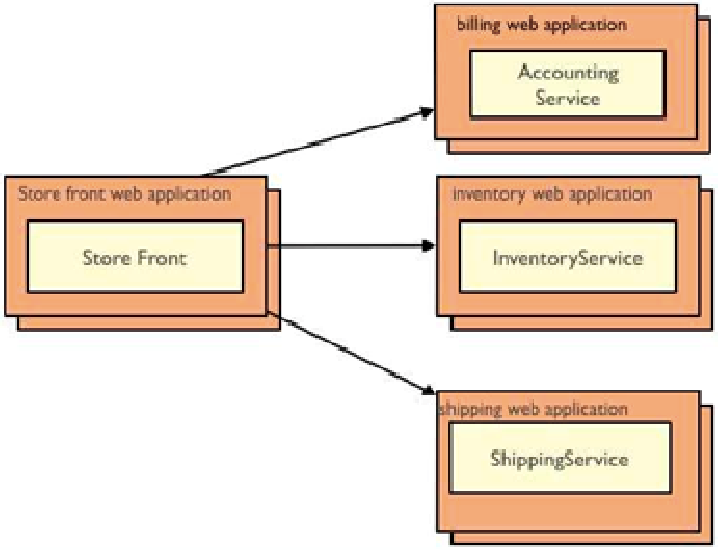
2.1.3 Modelo de arquitetura baseada em microsserviços

No contexto de microsserviços, as vantagens apresentadas na sessão anterior utilizando-se o modelo de arquitetura *web* tradicional são abordadas como uma série de inconvenientes. A grande base de código monolítico intimida os desenvolvedores devido à dificuldade de compreensão e modificabilidade do código fonte da aplicação como um todo. O *IDE,* devido à maior base de código, fica sobrecarregado. A dificuldade de escala eficiente é visível, visto que só pode ocorrer em uma dimensão tal que a arquitetura não pode vir a ser

[1] http://tomcat.apache.org

dimensionada com o volume crescente de dados e principalmente não é possível dimensionar cada componente de maneira independente.

Figura 3 - Aplicação utilizando arquitetura baseada em microsserviços



Fonte: Microservices architecture (2015).

A Figura 3 mostra uma aplicação que consiste em vários componentes que implementam a interface com o usuário como um conjunto de serviços. Cada serviço pode ser implantado de maneira independente dos demais serviços, tornando mais simples a adição de novas funcionalidades com maior frequência. E como os serviços são independentes o isolamento a falhas aumenta, pois o comportamento inadequado compreende somente o serviço afetado.

2.2 Aplicações monolíticas

Aplicações monolíticas são sistemas de *software* desenvolvidos com variados componentes e funcionalidades agrupados dentro de um grande sistema, fazendo desta uma aplicação projetada e construída em uma única unidade de *software*.

Em definição técnica, uma aplicação com abordagem de desenvolvimento monolítica possui diversos módulos em uma base de código unificada. Estes módulos são divididos entre funcionalidades técnicas e regras de negócio, e para sua execução frequentemente é efetuada uma compilação única para se construir a aplicação na íntegra, gerando assim um único binário executável ou implementável.

Em *Monolithic Design,* pela Cunningham & Cunningham (2014), são elencadas as seguintes dificuldades enfrentadas com o crescimento de aplicações monolíticas.

- A grande base de código monolítico torna o entendimento complexo.
- Dimensionamento da aplicação torna-se um desafio.
- Integração contínua e implantação tornam-se complexas e extensas.
- Sobrecarga da IDE em virtude da grande base de código.

Com os obstáculos pautados pela Cunningham & Cunningham (2014) durante o desenvolvimento crescente de uma aplicação monolítica, notam-se as dificuldades de transição de tecnologias, linguagem de programação ou *frameworks* em consequência da estrutura da aplicação estar intimamente interligada e dependente.

2.3 REST como modelo arquitetural

Criar e manter sistemas distribuídos naturalmente aparenta ser uma atividade complexa, pois envolve o desenvolvimento de artefatos que atendam desde os requisitos de negócio até os requisitos da aplicação, como o gerenciamento das transações distribuídas e os protocolos de comunicação, que serão elucidados posteriormente na sessão 4.4 a respeito da comunicação dos microsserviços.

A utilização de boas práticas de desenvolvimento tende a reduzir o tempo necessário para se construir um sistema como um todo; no sentido de desenvolver uma aplicação com processamento distribuído e interconectado, o REST (*Representational State Transfer*) pode ser utilizado como um modelo arquitetural capaz de atender aos requisitos complexos, sem agregar complexidade e com alto nível de acoplamento na integração dos microsserviços, mesmo que estrategicamente os modelos de domínio sejam projetados para uma arquitetura neutra.

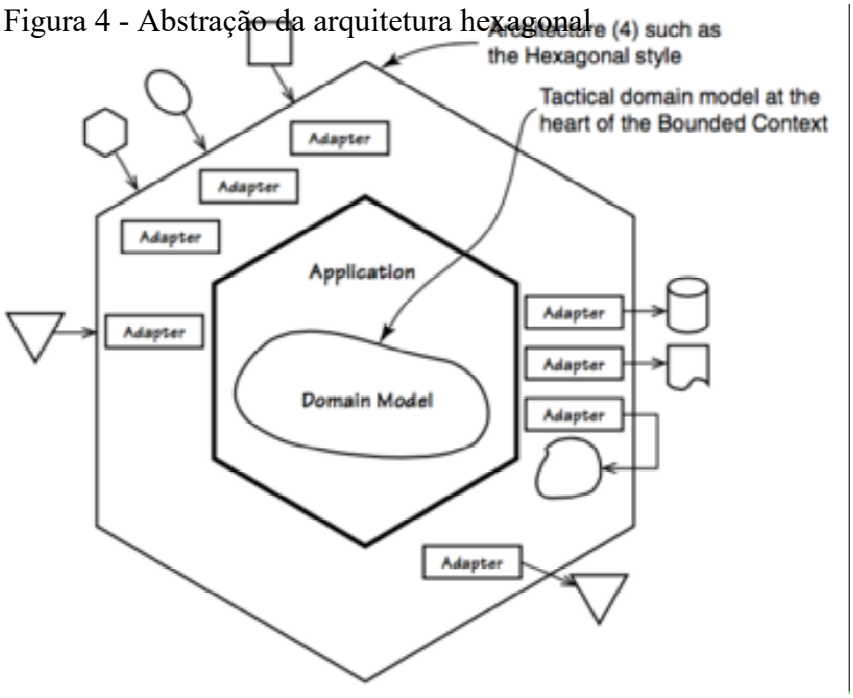
Fielding (2000) defende a utilização do REST para interligar sistemas essencialmente heterogêneos, sendo possível realizar a troca de mensagens e informações mantendo a semântica dos dados entre os dispositivos envolvidos, e ainda garantir a segurança, integridade e consistência nos dados distribuídos.

Vernon (2013), por demasiada ênfase na arquitetura, defende um modelo chamado de Hexagonal, que pode ser utilizado para facilitar a compreensão e o desenvolvimento do modelo REST e orientação a eventos, focando na importância de elaborar cuidadosamente um

modelo baseado em DDD ou *Domain Driven Design*, conceito melhor explicitado posteriormente na sessão 2.4.1 Design orientado a domínio.

A Figura 4, em seguida, descreve uma abstração de arquitetura hexagonal, definindo o modelo de domínio como o centro do *software*, com sua interface bem definida e projetada para o consumo dos serviços por clientes distintos, relativamente simples de se empregar e essencialmente focada no modelo de domínio.

Figura 4 - Abstração da arquitetura hexagonal



Fonte: VERNON (2013).

2.4 Componentização dos microsserviços

Tradicionalmente, um componente é uma unidade de *software* independente que pode ser representada como substituível e atualizável. Bibliotecas de *software* usualmente são componentes que estão diretamente ligados a um programa. No caso dos serviços, estes são componentes fora do processo, e, para sua comunicação, deve-se utilizar chamadas de procedimentos remotos, por esta razão serviços independentes devem ser considerados componentes em vez de bibliotecas ou outras classificações.

O planejamento prévio para se projetar e construir as interfaces de serviços em desenvolvimento é uma vantagem desta abordagem, visto que qualquer desenvolvimento sem o planejamento adequado da componentização tende a resultar em um código não sustentável.

Salientando a componentização dos microsserviços, segundo Namiot e Sneps-Sneppe (2014), deve-se utilizar as seguintes premissas para o desenvolvimento preciso de sistemas baseados em microsserviços:

- Chamadas/Respostas devem utilizar dados arbitrariamente estruturados.
- Eventos assíncronos devem fluir em tempo real e em ambas direções.
- Pedidos e respostas podem fluir em qualquer direção.
- Pedidos e respostas podem ser arbitrariamente aninhados.
- O formato de serialização de mensagem deve ser conectável (JSON, XML).

2.4.1 *Design* orientado a domínio

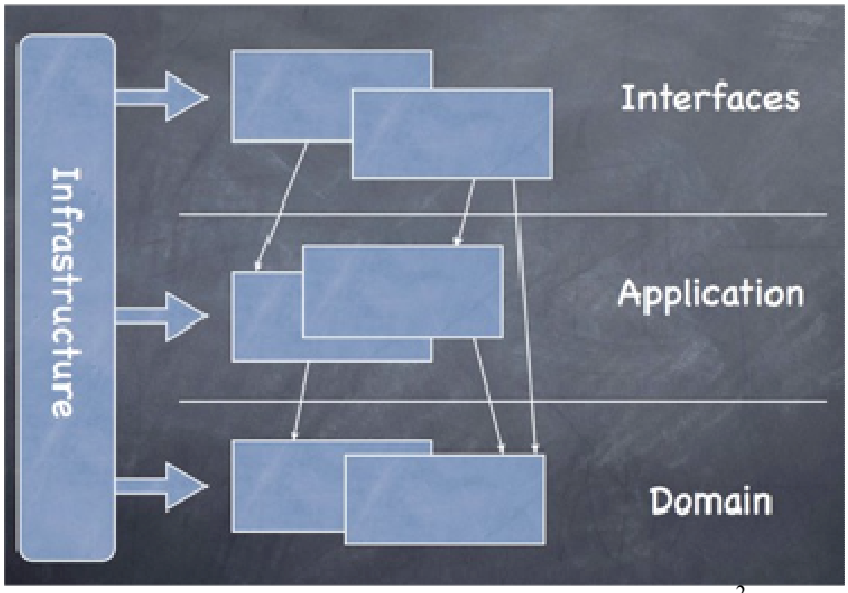
Evans (2015) ajuda a entender a importância de se representar as características do mundo real em código, visto que o *Domain Design Driven* ou DDD é um tema amplo e que contém detalhes de difícil incorporação em uma base de código. Sendo assim, deve-se considerar os conceitos em que a lógica de negócio ou partes do domínio geralmente possam ficar escondidas. O intuito do DDD está em escrever um código menos acoplado e mais coeso, e, como consequência, facilitar sua compreensão mesmo em situações de grande escala.

Segundo Vernon (2013), no DDD o foco está na lógica de negócios e no domínio. Isso leva a desenvolver maneiras melhores e mais estruturadas de comunicação e interação entre máquinas.

Com a constante evolução dos recursos computacionais a compreensão das plataformas de virtualização permite dispor e redimensionar máquinas segundo uma necessidade específica, com uma infraestrutura que disponibiliza diversas maneiras de lidar com os problemas de grande escala.

Abordando o DDD no contexto da responsabilidade única e arquitetura limpa, Martin (2003) define responsabilidade como uma razão e eixo para mudança. Quando se possui uma classe, e é possível pensar em mais de um motivo para se realizar uma mudança, então bem provavelmente essa mesma classe possuirá mais de uma responsabilidade. Quando um requisito tende a mudar, essa mudança se manifesta principalmente através da alteração da responsabilidade entre classes, desse modo, se uma classe assume mais de uma responsabilidade haverá mais de uma razão para que ela mude. Segundo Martin (2003), desenvolvedores se habituam a pensar em responsabilidades em grupos.

Figura 5 - Ilustração do modelo de camadas em intercomunicação



Fonte: Modelo tradicional de divisão de *software* por camadas[2]

Na Figura 5, a camada *Interfaces* apresenta e interpreta os comandos oriundos do usuário, a camada *Application* coordena as atividades da aplicação não contendo as lógicas de negócio, a camada *Domain* contém as informações principais do domínio e a camada *Infrastructure* suporta as demais camadas, assegura a comunicação entre elas e ainda implementa a persistência.

2.5 Computação em Nuvem

O conceito de computação em nuvem se refere à utilização de computadores compartilhados e interligados por meio da internet. Segundo Marinescu (2013), a computação em nuvem é um movimento iniciado nas primeiras décadas do novo milênio, sendo motivado pela ideia de que o processamento de informações pode ser feito de forma mais eficiente em grandes centros de sistemas de computação e armazenamento acessíveis através da internet.

Para Marinescu (2013), a computação em nuvem é uma evolução do paradigma da computação local, em que aplicações científicas, mineração de dados, jogos, redes sociais, dentre outras inúmeras atividades computacionais que fazem o uso intensivo de dados são visivelmente beneficiadas pelo armazenamento de informações na nuvem.

A computação em nuvem deve oferecer serviços de computação e armazenamento escaláveis e estáticos de maneira transparente para o usuário final, e a utilização dos recursos

[2] http://dddsample.sourceforge.net/architecture.html

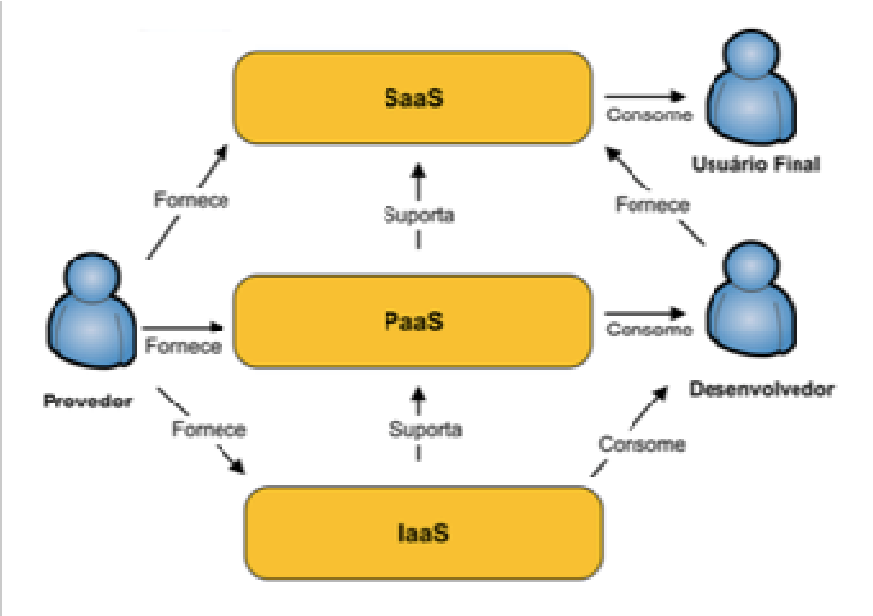
pode ser facilmente medida. Os recursos de TI são fornecidos como um serviço, permitindo ao usuário final consumi-los sem a necessidade de conhecimento sobre a tecnologia utilizada.

Marinescu (2013) ainda esclarece que na computação em nuvem a operação é mais eficiente e o aumento de confiabilidade e segurança também é maior, baseando-se na multiplexação de recursos e principalmente devido à economia em escala, configurando-se como uma realidade técnica e social dos dias atuais, e, ao mesmo tempo, uma tecnologia emergente.

Pode-se utilizar os três tipos de classificação como expostos na Figura 6, ou seja, papeis na computação em nuvem para facilitar o entendimento da tecnologia:

- **SaaS:** *Software as a Service* ou *Software* como serviço.
  O modelo de SaaS proporciona sistemas de *software* com propósitos específicos, que estão disponíveis para o usuário através da internet. Os sistemas são acessíveis a partir dos vários dispositivos do usuário.
- **PaaS:** *Plataform as a Service* ou Plataforma como serviço.
  O PaaS oferece uma infraestrutura de alto nível de integração para implementar e testar aplicações na nuvem. O usuário não administra ou controla a infraestrutura, mas tem controle sobre as aplicações implantadas.
- **IaaS:** *Infrastructure as a Service* ou Infraestrutura como serviço.
  A Infraestrutura como serviço é a parte responsável por prover toda infraestrutura para o PaaS e SaaS, com o objetivo principal de tornar mais acessível o fornecimento de recursos de computação fundamentais para se construir um ambiente sob demanda. O termo IaaS se refere a uma infraestrutura computacional baseada em técnicas de virtualização de recursos de computação.

Figura 6 - Papéis na computação em nuvem



Fonte: SOUSA, MOREIRA, MACHADO (2009).

A computação em nuvem reforça a ideia de que a computação e a comunicação estão fortemente entrelaçadas. Uma das desvantagens da computação em nuvem no contexto de microsserviços em ambientes de redes locais é que ela pode não emergir como uma possível alternativa aos paradigmas tradicionais para aplicações com intensivo acesso a dados por meio da internet em que os principais gargalos são largura de banda, latência e custo de comunicação.

2.5.1 *Uniform Resource Locator*

A URL ou *Uniform Resource Locator* refere-se ao endereço de rede onde se encontra algum recurso computacional. Conforme Berners-Lee et al. (1998, p. 1, destaques do autor) essa rede pode ser desde a internet como qualquer rede computacional corporativa:

> The specification is derived from concepts introduced by the World Wide Web global information initiative, whose use of such objects dates from 1990 and is described in "Universal Resource Identifiers in WWW", RFC 1630. The specification of URLs is designed to meet the requirements laid out in "Functional Requirements for Internet Resource Locators"

É utilizando o endereço URL que se obtém acesso aos recursos projetados e disponibilizados pelo microsserviços por intermédio de uma API.

2.6 Arquitetura de *Software*

O *Working Group on Architecture* (1998) da IEEE define arquitetura como "o conceito de nível mais alto de um sistema em seu ambiente". Este padrão proposto pelo grupo recomenda uma descrição da arquitetura com base no conceito de múltiplas visões. A finalidade das múltiplas visões neste cenário é apoiar a compreensão do *software* utilizando o vasto conjunto de tipos de dados e auxiliar na análise do desempenho da comunicação utilizada através da rede de computadores interligados.

Apesar da dificuldade em definir arquitetura de *software* com precisão esta se configura como um aspecto do *design* de *software* que se concentra na adequação e integridade do sistema e nas preocupações estéticas, levando em consideração o sistema como um todo no ambiente de desenvolvimento e do usuário, organizando e estruturando os componentes significativos do sistema.

Garlan e Shaw (1993) sugerem que a arquitetura de *software* é um nível de *design* voltado para questões que vão além dos algoritmos e das estruturas de dados da computação. A projeção e a especificação da estrutura geral de um sistema emergem como um novo tipo de problema e as questões estruturais incluem organização total, estruturas de controle globais, protocolos de comunicação, sincronização e acesso a dados, atribuições de funcionalidades a elementos de *design*, distribuição física, escalonamento e desempenho.

A arquitetura de *software* é resultado do fluxo de trabalho no desenvolvimento de *software*, estando em constante desenvolvimento, refinamento e aprimoramento.

2.7 Resiliência

Segundo Newman (2015), o conceito principal de engenharia de resiliência é o *bulkhead* ou resguardo. Se algum componente do sistema falhar é possível isolar o problema e os demais sistemas ainda continuam operacionais. Em um sistema monolítico, se algum dos serviços falhar o sistema em sua completude é interrompido, já em um sistema monolítico que pode ser executado em várias máquinas pode-se reduzir a chance de falhas, no entanto, com os microsserviços pode-se construir sistemas que lidem com o fracasso total dos serviços, sem degrader as funcionalidades em conformidade com os requisitos.

2.8 *Scrum*

O *Scrum* é uma metodologia ágil e um *framework* para desenvolvimento iterativo e incremental com foco na gestão e planejamento de projetos de *software*. Os projetos no *Scrum* são divididos em ciclos e são nomeados de *Sprint*, sendo que elas representam um conjunto de atividades a serem executadas. Além disso o *Scrum* é um conjunto de princípios e práticas que fornece uma base para que se utilize praticas de engenharia de *software* relevantes para as atividades de construção de sistemas.

3 TRABALHOS RELACIONADOS

Esta seção descreve os trabalhos relacionados que se instituem como incito e referência para o enriquecimento desta proposta.

3.1 *On Micro-services Architecture*

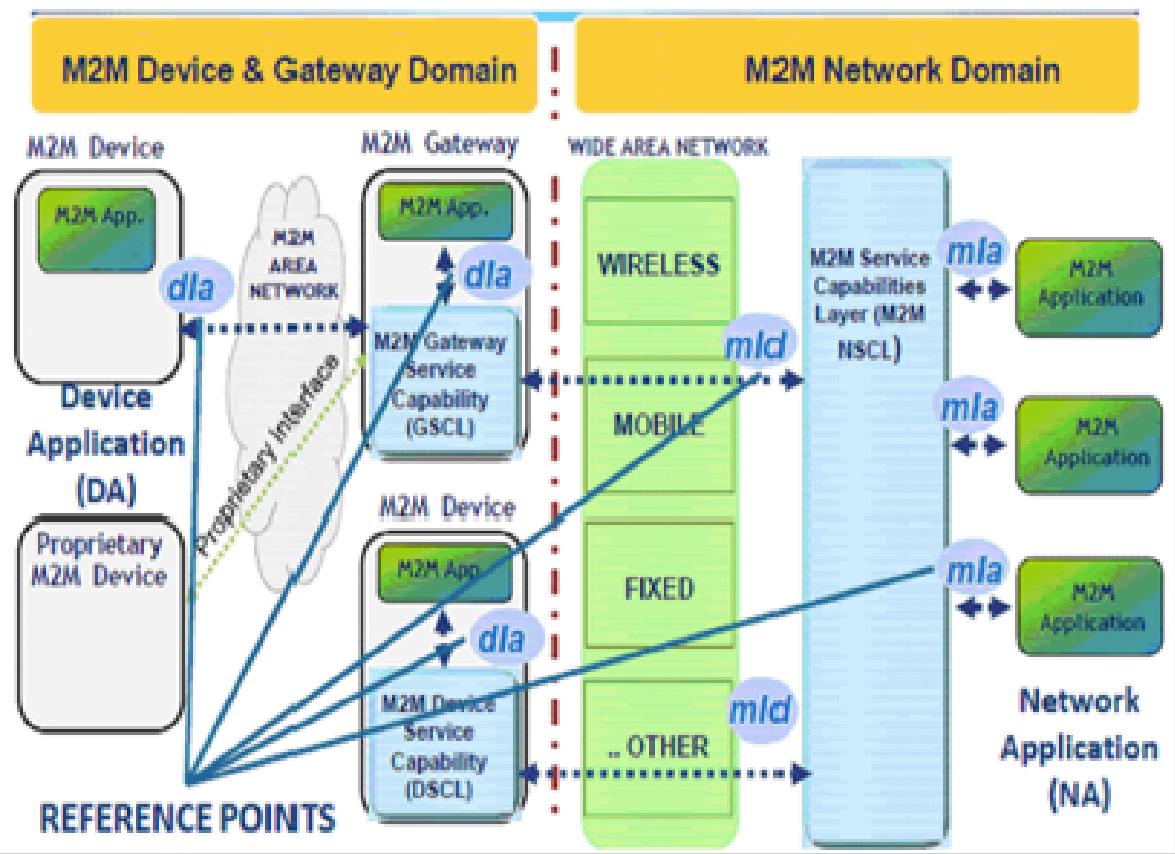
Namiot e Sneps-Sneppe (2014) descrevem a implementação e a visão geral de uma arquitetura baseada em microsserviços, abordando o fato de que o desenvolvimento deve tornar os serviços leves, independentes e executores de funções simples colaborando entre si por meio de interfaces bem definidas. Foram identificados os princípios comuns utilizando-se esta abordagem, sendo possível avaliar as vantagens e desvantagens do uso da arquitetura de microsserviços e como foi a transição natural do desenvolvimento de aplicações utilizando o *framework Machine to Machine* (M2M), considerado parte integral do conceito da internet das coisas ou *Internet of Things* (IoT).

Esse trabalho também apresenta algumas limitações do uso dos microsserviços. Os autores supracitados destacam que na prática esta abordagem possui seu próprio conjunto de inconvenientes, tais como a complexidade adicional para se planejar e criar sistemas distribuídos, tornando os testes de *software* mais complexos e dificultando principalmente a implementação do mecanismo de comunicação interserviços e de transações distribuídas.

Além disso, os microsserviços contribuem para o aumento do consumo de memória, devido aos independentes espaços de memória alocados para cada um dos serviços. Outro desafio citado é como dividir ou particionar o sistema em microsserviços. Uma abordagem planejada pelos autores foi particionar os serviços em casos de uso, que serão exemplificados posteriormente na Figura 8, seguindo os princípios definidos pelo M2M ETSI, mostrado a seguir na Figura 7.

Na Figura 7, o conceito *Machine to Machine,* ou máquina à máquina, refere-se à tecnologia que permite que sistemas se comuniquem com outros sistemas ou dispositivos que também possuam a mesma habilidade. A comunicação se dá originalmente através de uma rede de comunicação, assim, as informações são transmitidas e analisadas por uma tecnologia centralizadora e, posteriormente, podem ser roteadas para um sistema computacional.

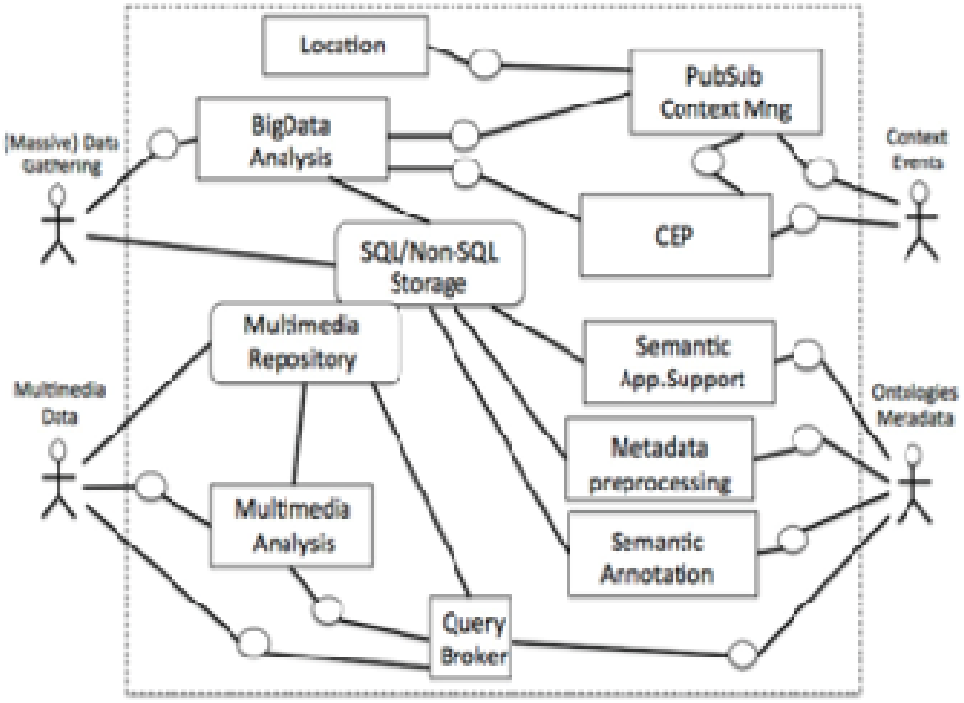
Figura 7 - Modelo *European Telecommunications Standards Institute* M2M



Fonte: ETSI M2M (2014).

Outra estratégia é o particionamento por verbos e substantivos, exemplificado na Figura 8, em que serviços poderiam implementar subsistemas como por exemplo *login* e *backup*, enquanto o particionamento por substantivos trataria como recursos os serviços responsáveis por operarem em entidades de determinados tipos. O ideal é que cada serviço possua um pequeno conjunto de responsabilidades e siga o padrão *Single Responsibility Principle* ou Princípio da Responsabilidade Única.

Figura 8 - *FI-WARE data model*, particionado em verbos e substantivos



Fonte: Elmangoush at al. (2013).

De forma semelhante, a abordagem proposta neste trabalho de conclusão de curso também utiliza uma prática de desenvolvimento focada em sistemas distribuídos com o emprego de testes unitários e um conceito de Contêiner, posteriormente explanado na sessão específica 4.6 para escalar os microsserviços da aplicação, além de também implementar um mecanismo de comunicação de interserviços e utilizar a abordagem de particionamento em verbos para extrair funcionalidades com responsabilidades específicas impostas por limites explícitos.

3.2 *Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications*

Viennot et al. (2015) apresentam o *Synapse*, um sistema de código fonte aberto, fortemente semântico, baseado em uma abordagem de integração e replicação de serviços *web* orientado a dados para grande escala. Este é um sistema de replicação *cross-DB* que simplifica o desenvolvimento e evolução de aplicações *web* baseadas em dados. Os microsserviços são independentes e compartilham dados entre si de maneira isolada e escalável, atuando sobre os mesmos dados, desse modo, cada serviço é executado com sua própria base de dados; os SGBD ou Sistema Gerenciador de Banco de Dados pode ser distinto, porém, incorpora as visões

compartilhadas entre serviços. Isso permite que os bancos de dados operem em diferentes esquemas, com motores distintos.

O sistema sincroniza em tempo real os subconjuntos de dados de maneira transparente, utilizando um mecanismo de replicação consistente e escalável, aproveitando os modelos de dados de alto nível e estendendo a arquitetura MVC, em que os desenvolvedores separam logicamente os modelos da aplicação *web* para realizarem a replicação entre os bancos de dados heterogêneos por meio da distribuição horizontal dos dados. Os controladores definem as unidades básicas em que os dados são lidos, manipulados e escritos, implementando a lógica de negócios e agindo sobre os dados. Através da API é possível especificar quais os dados serão compartilhados entre os serviços, além disso, os modelos são expressos por objetos de alto nível, mapeados automaticamente via ORM ou *Object-relational mapping*.

O *Synapse* foi desenvolvido utilizando o *framework Ruby on Rails*, suportando replicação e propagação de dados entre uma ampla variedade de sistemas de gerenciamento de banco de dados com suporte na linguagem de consulta estruturada ou *Structured Query Language* (SQL) e NoSQL, incluindo MySQL, PostgreSQL, Oracle, MongoDB, Cassandra, entre outros, como pode ser visto na Tabela 1.

Tabela 1 – Bancos de dados suportados pelo *Synapse*

Type	***Supported vendors***	***Example use cases***
Relational	*PostgreSQL, MySQL, Oracle*	*Highly structured content*
Document	*MongoDB, TokuMX, RethinkDB*	*General purpose*
Columnar	*Cassandra*	*Write-intensive workloads*
Search	*Elasticsearch*	*Aggregations and analytics*
Graph	*Neo4j*	*Social network modelink*

Fonte: VIENNOT et al. (2015).

Nele foi implementado um conjunto de microsserviços que servirão como referência em tecnologia de desenvolvimento e linguagem de programação para a base do método proposto neste trabalho e aplicação no estudo de caso de desenvolvimento do *software* de dosagem para o tratamento com aplicação de fototerapia. O *Synapse* está atualmente executando em produção para mais de 450.000 usuários, por mais de dois anos. Ele simplifica a construção e evolução de aplicações *web* orientadas a dados complexos, proporcionando um alto nível de agilidade, crucial para aplicações que utilizam grandes fontes de dados em expansão.

De maneira semelhante, este projeto de pesquisa estende a arquitetura MVC para o desenvolvimento em camadas dos microsserviços, e, caso necessário, cada serviço dispõe de um mecanismo de persistência que melhor se adapta ao modelo de dados que este gerencia e compartilha através de uma API de alto nível.

3.3 *Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems*

Levcovitz et al. (2015) propõem uma técnica para extração de microsserviços a partir de sistemas monolíticos, seguindo o conceito de construção de uma aplicação complexa como um conjunto de serviços pequenos, coesos e independentes. Esta técnica auxilia na identificação e definição dos microsserviços tendo como base um sistema monolítico, identificando assim bons candidatos para serem transformados em microsserviços, reduzindo o tamanho do sistema original e transportando os benefícios da arquitetura de microsserviços.

Segundo Richardson (2014), sistemas monolíticos ficam maiores ao longo do tempo, desviando-se da arquitetura projetada, tornando sua evolução complexa, arriscada e de custo elevado.

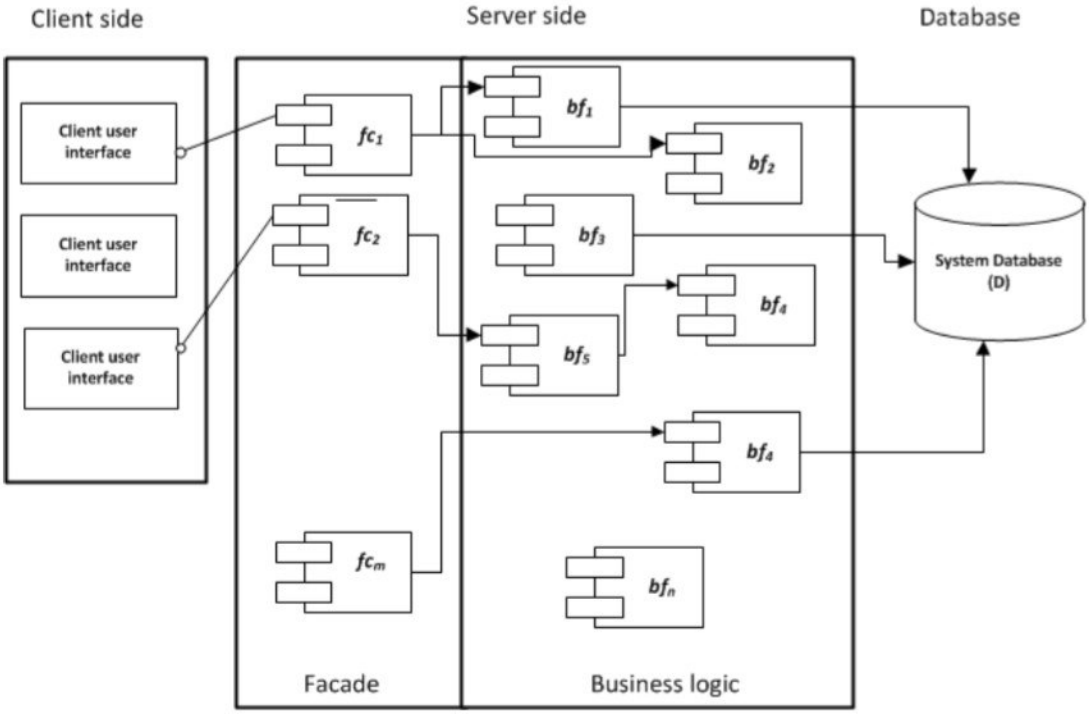
Nesse artigo de Levcovitz et al. (2015), foi aplicada a técnica de extração de microsserviços em um sistema monolítico bancário de 750 KLOC, ou seja, 750 mil linhas de código, que gerencia 3,5 milhões de contas bancárias, realizando quase 2 milhões de autorizações por dia e se mostra uma alternativa promissora para modernização empresarial tendo como base sistemas monolíticos legados.

A técnica considera que aplicações monolíticas possuem 3 partes principais, posteriormente aprofundadas na sessão 2.1 “Microsserviços”:

- Interface de usuário do lado do cliente.
- Um aplicativo do lado do servidor.
- Um banco de dados.

Como detalhado na Figura 9, em seguida, é considerado que o sistema como um todo está estruturado em subsistemas menores e cada subsistema possui um conjunto bem definido de responsabilidades.

Figura 9 - Aplicação monolítica



Fonte: Levcovitz et al. (2015)

Em termos formais, assume-se que o sistema S é representado por uma tripla (F, B, D), onde F = {fc_1, fc_2, ..., $fc_{n'}$} é um conjunto de fachadas, B = {bf_1, bf_2, ..., $bf_{n''}$} é um conjunto de funções de negócio, e D = {tb_1, tb_2, ..., $tb_{n'''}$} é um conjunto de tabelas do banco de dados. As fachadas (fc_i) são pontos de entrada que invocam funções de negócio (bf_i), as funções de negócio são métodos que codificam as regras de negócio e dependem das tabelas do banco de dados (tb_i). Também é importante definir uma unidade organizacional O = {a_1, a_2, ..., a_n} dividida em áreas responsáveis pelos processos de negócio.

A técnica para identificar microsserviços em um sistema S consiste em 6 etapas, sintetizadas a seguir.

Etapa #1: Mapear as tabelas do banco de dados $tb_i \in D$ em subsistemas $ss_i \in SS$. Cada subsistema representa uma área de negócio (a_i) da organização.

Etapa #2: Criar um gráfico de dependência (V, E), onde os vértices representam fachadas ($fc_i \in F$), funções de negócio ($bf_i \in B$), ou tabelas do banco de dados ($tb_i \in D$), e as arestas podem representar: chamadas das fachadas para funções de negócio, chamadas entre funções de negócio e acesso entre funções de negócio para tabelas do banco de dados.

Etapa #3: Identificar pares (fc_i, tb_j), onde $fc_i \in F$ e $tb_j \in D$, não havendo caminho a partir de fc_i para tb_j no gráfico de dependências.

Etapa #4: Para cada subsistema ss_i previamente definido no Passo #1, selecione pares (fc_i, tb_j) identificados na etapa anterior, onde $tb_j \in ss_i$.

Etapa #5: Identificar candidatos a serem transformados em microsserviços. Para cada par distinto (fc_i, tb_j) obtido na etapa anterior deve-se inspecionar os códigos da fachada e das funções de negócio no caminho do vértice fc_i para tb_i no gráfico de dependências. Essa estratégia tem como objetivo identificar quais regras de negócio realmente dependem das tabelas do banco de dados e as operações devem ser descritas em forma de regras, com objetivos, entrada/saída, funcionalidades e dados.

Etapa #6: Criar a API para transformar a migração dos microsserviços transparentes para os clientes. A API consiste em uma camada intermediária entre cliente e servidor, utilizando as mesmas tecnologias e interface como fc_i, que lida com as solicitações do lado do cliente, sincroniza chamadas para o novo microsserviço M e fc'_i, uma nova versão de fc_i sem o código extraído e implementado no microsserviço. Para cada fachada deve ser definida uma API.

Traçando um paralelo com este trabalho de conclusão de curso, este método contribui viavelmente para avaliar sistemas, identificar e descrever microsserviços, inclusive diferenciar subsistemas que não são bons candidatos para migrar para o desenvolvimento baseado em microsserviços, além de auxiliar na identificação de cenários e isolamento de fragmentos que requerem um esforço adicional para migrar subsistemas para um conjunto de microsserviços.

O próximo capítulo descreve os conceitos, decisões e abordagens, relacionados por meio dos procedimentos metodológicos, que compreendem a elaboração deste projeto de pesquisa.

4 PROCEDIMENTOS METODOLÓGICOS

Primeiramente, realizou-se uma revisão bibliográfica para identificação dos modelos de desenvolvimento baseados em microsserviços existentes. Identificou-se as características e os princípios utilizados para o desenvolvimento de *software* baseado em microsserviços e estudou-se ferramentas e tecnologias como suporte ao desenvolvimento baseado em microsserviços. Com esses conhecimentos, propôs-se um método de desenvolvimento de *software* no contexto de microsserviços e, por fim, realizou-se um estudo de caso utilizando o método proposto.

4.1 Revisão bibliográfica

Este trabalho de conclusão de curso se iniciou com a fase de revisão bibliográfica, cujo foco foi o desenvolvimento de competências sobre os seguintes temas:

- Microsserviços.
- Computação em Nuvem.
- Virtualização.

4.2 Estruturar logicamente os artefatos da solução

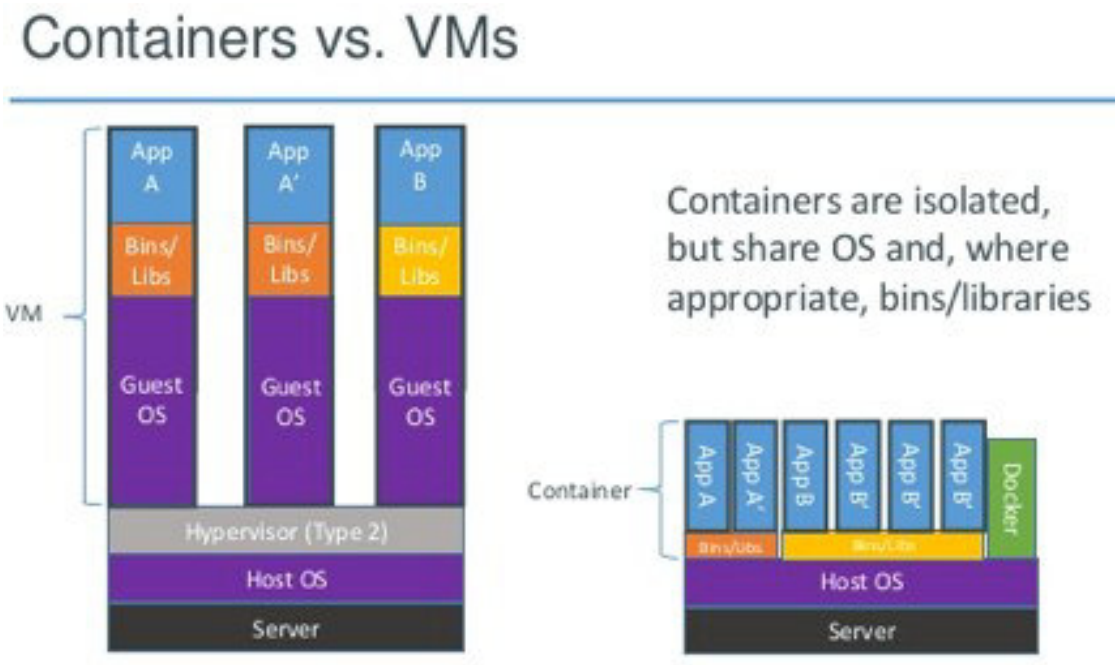
O planejamento e a definição hierárquica da árvore de diretórios da solução têm como intuito facilitar a organização lógica dos artefatos gerados antes, durante e após os processos que permeiam o desenvolvimento e a entrega do *software*, enfocando como objetivo principal a separação de interesses e a abstração da alocação de espaço físico pelo computador. O sistema de arquivos é a estrutura lógica utilizada pelo sistema operacional que será a base para modularização dos microsserviços desenvolvidos.

4.3 Definir tecnologias para auxiliar a construção de microsserviços

Com a constante evolução dos recursos computacionais, a compreensão das plataformas de virtualização e tecnologias de contêiner permite dispor e redimensionar recursos físicos e lógicos segundo a necessidade, através de uma infraestrutura que disponibilize diversas maneiras de lidar com o problema da escalabilidade e distribuição de recursos computacionais.

A seguir, é apresentada uma comparação visual entre as tecnologias de contêiner e máquinas virtuais, separando em cores os recursos distintos, porém, dedicados à aplicação, sistema operacional e *hardware* ou computador, comumente presentes em tais tecnologias.

Figura 10 - Tecnologia de virtualização tradicional e contêiner



Fonte: *VMware buys into Docker containers*, diferenças entre virtualização e contêiner

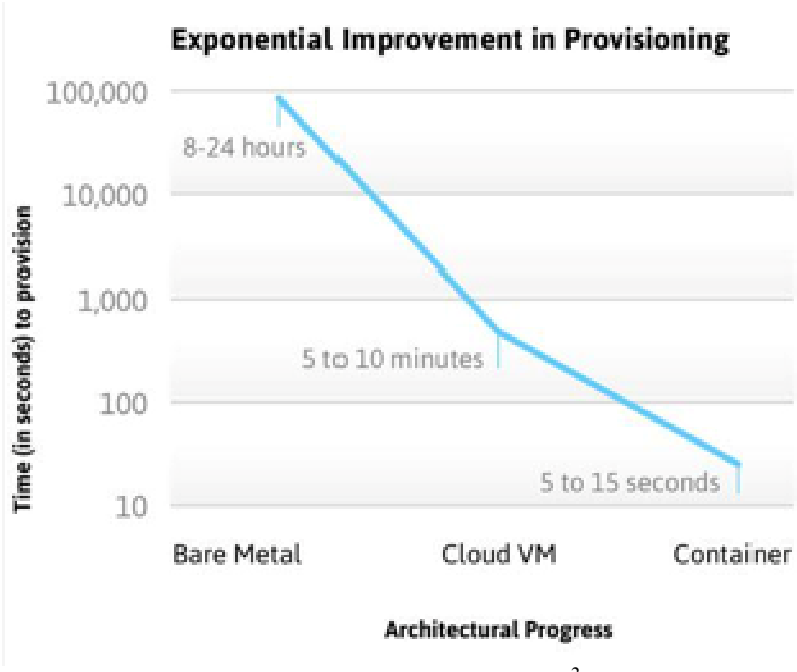
Como pôde ser visto na Figura 10, as diferenças entre as tecnologias tradicionais de máquina virtual ou *virtual machine* (VM), virtualização e paravirtualização para contêiner, são aparentes, sendo que na VM, além das aplicações e dependências, é necessária a execução de um sistema operacional (SO) completo, enquanto o contêiner compreende apenas os aplicativos e suas dependências para executar o *software* em completude.

As máquinas virtuais padronizadas são capazes de dividir e distribuir recursos computacionais de maneira viável, sem depender do *kernel* do sistema operacional e até utilizando *hardware* separado. Porém, executar um SO separadamente para obter recursos e isolar a segurança, desconsiderando a inicialização lenta devido à espera pelo próprio SO, leva a um desperdício de recursos imprescindíveis para a execução da aplicação, com o consumo maior, geralmente por parte do SO de memória de acesso aleatório (RAM) e disco rígido (HD), do que o expectável para as próprias aplicações hospedadas.

A Figura 11, a seguir, mostra a comparação em tempo na escala de milissegundos, com base no progresso arquitetural, entre o desacoplamento do provisionamento para a

implantação de *hardware* por parte de uma máquina virtual e o desacoplamento do provisionamento do sistema operacional e inicialização (boot) utilizando uma tecnologia de contêiner. O *Bare Metal* também é uma forma de virtualização de mais baixo nível, conhecida como Tipo 1, em que o sistema operacional se comunica diretamente com o *hardware*.

Figura 11 - Provisionamento das tecnologias de VM tradicionais e contêineres



Fonte: Melhoria no provisionamento de recursos[3]

4.4 Planejar a comunicação dos microsserviços

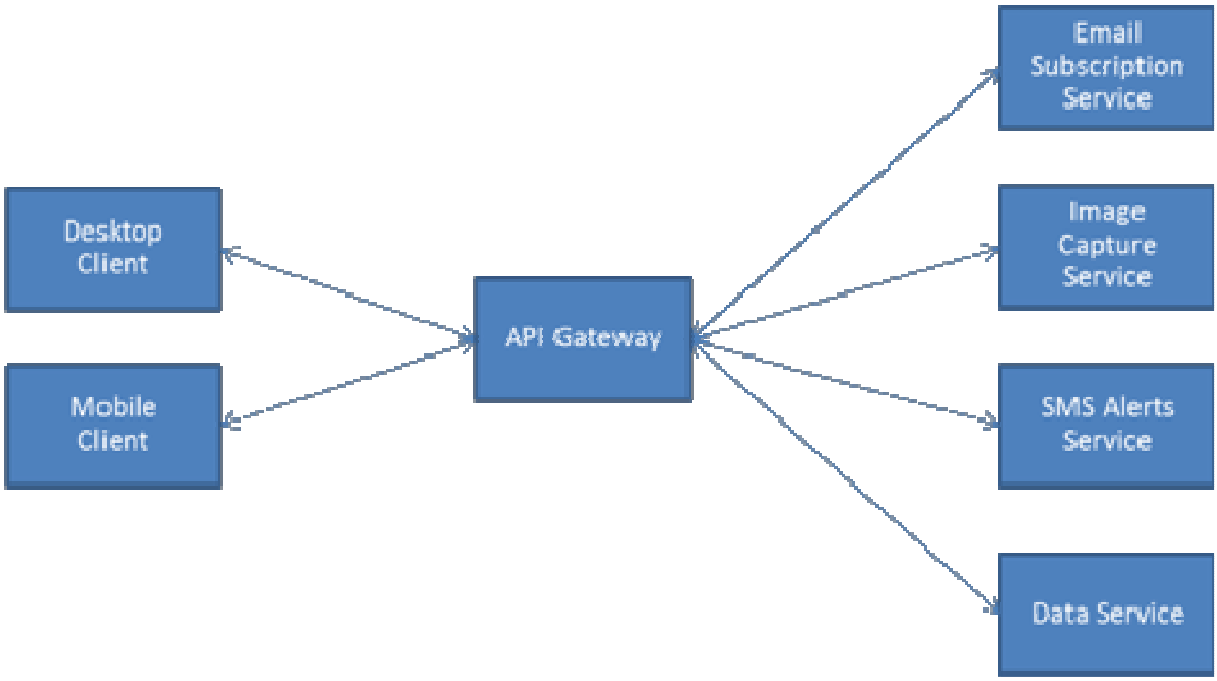
Uma análise cuidadosa de como os clientes se comunicam com os serviços é de extrema relevância. Em um sistema baseado em microsserviços uma consideração importante no projeto está na comunicação do cliente com os microsserviços, desse modo, como os serviços devem ser processos independentes e pequenos, considerar e planejar as possibilidades específicas de como os clientes deverão se comunicar deve ser uma tarefa crucial e prioritária.

Uma abordagem a se adotar é permitir que os clientes se comuniquem apenas com um serviço nomeado de *API Gateway*, que atua como um receptor agregador de diferentes serviços, com o propósito de receber as solicitações por parte dos clientes.

No diagrama apresentado na Figura 12, um *API Gateway* é posicionado entre as aplicações do cliente e os microsserviços, oferecendo uma API adaptada e fortemente semântica aos diferentes tipos de aplicações que irão consumi-la.

[3] http://www.linuxjournal.com/content/containers—not-virtual-machines—are-future-cloud?page=0,1

Figura 12 - Uma abordagem para comunicação dos microsserviços



Fonte: Abordagem para comunicação centralizada dos microsserviços[4]

Para facilitar o entendimento da comunicação entre microsserviços é necessário entender a diferença entre sistemas baseados em processos e eventos. Compreender esse propósito nos permite preconceber a metodologia que mais se adequa à construção de um sistema baseado em microsserviços.

Segundo Balla (2014), nos sistemas fundamentados em processos, um único processo é dedicado a servir a uma única solicitação. Caso a solicitação seja sobre a obtenção de alguns dados de um armazenamento permanente, ou outro serviço, o processo será bloqueado à espera da operação de entrada para ser concluído. Em sistemas baseados em eventos, há um único processo, sendo que os eventos podem ser processados em threads distintas para servir a um grande número de pedidos, não bloqueando as operações de entrada.

A seguir são elencadas as características básicas principais, citadas por Balla (2014), de sistemas baseados em processos e em eventos.

- Sistemas baseados em Processos:
  - Processo simples / *Threads* por pedido.
  - I/O bloqueante.
  - Conjunto encadeado de processos.

[4] http://www.developer.com/open/building-microservices-with-open-source-technologies.html

- Sistemas baseados em Eventos:
    - Processo único para grande número de pedidos.
    - I/O não bloqueante.
    - Usa um processo por *Core* em sistemas escaláveis.

Entender o propósito e suas diferenças permite adotar o estilo correto para planejar a comunicação dos microsserviços. Balla (2014) auxilia a metodologia e proposta de desenvolvimento esclarecendo que um sistema baseado em processos é a opção ideal quando se possui um alto poder computacional, já um sistema baseado em eventos é ideal para a utilização intensiva de recursos de Entrada e Saída (I/O ou *input-output*) e gerenciamento de dados.

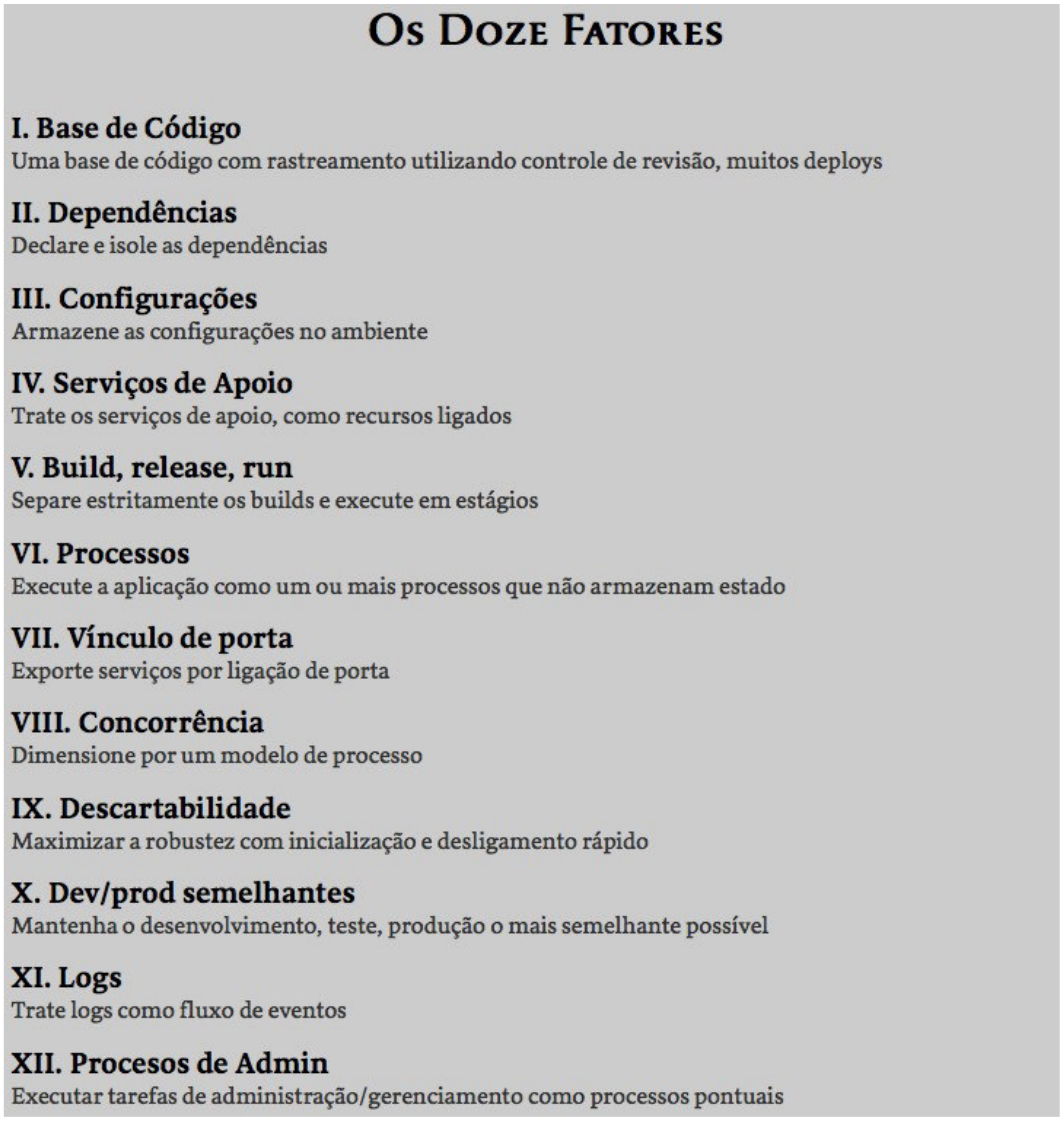
4.5 Definir método para construir microsserviços

Esta etapa consistiu em definir o método para a construção dos microsserviços. Para tanto, este trabalho se baseou no manifesto denominado *The Twelve-Factor App* que disponibiliza uma abordagem metodológica para construção de serviços de *software*, defendendo as seguintes premissas:

- Utilizar formatos declarativos para automação de configuração.
- Possuir contrato (*interface*) bem definido com o sistema subjacente.
- Ser adequado para a implantação em plataformas de nuvem modernas.
- Minimizar a divergência entre desenvolvimento e produção.
- Possibilitar a escalabilidade vertical.

Os contribuintes deste manifesto estão diretamente envolvidos no processo de desenvolvimento, escala e implantação de centenas de aplicativos, sintetizando a experiência dos observadores sobre uma ampla variedade de *softwares* entregues como um serviço, triangulando as práticas de desenvolvimento, com atenção especial para a dinâmica do crescimento orgânico do *software* ao longo do tempo e também para a colaboração entre os desenvolvedores.

Figura 13 - Princípios apoiados pelo *The Twelve-Factor* ou Os Doze Fatores



Fonte: *The Twelve-Factor App*

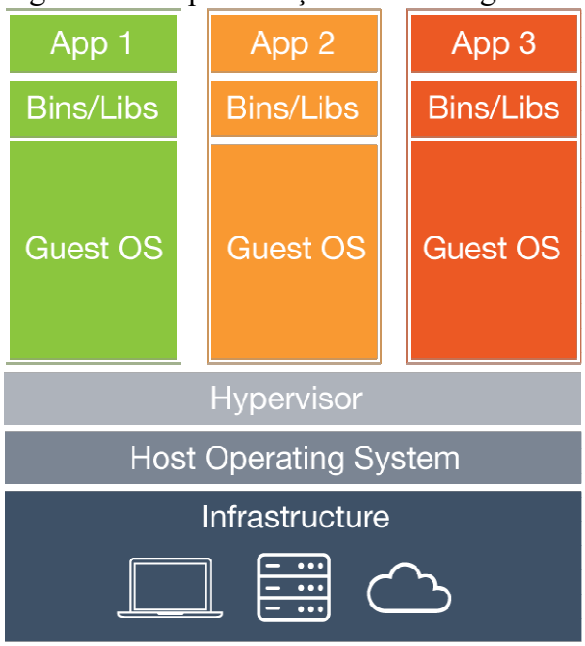
Como segue na Figura 13, esta referência de metodologia pode ser aplicada em *softwares* escritos em linguagens de programação diferentes e que utilizarão quaisquer combinações de serviços de apoio, como banco de dados, filas, cache de memória e etc., conforme explicitado detalhadamente na sessão 5.3 “Definir relacionamento entre as funcionalidades”. O foco do manifesto está em desenvolvedores e engenheiros que implantam ou administram *software* como um serviço, abordando assuntos como controle de versão, ambiente, construção, comunicação e execução de tarefas de gestão e administração.

4.6 Implantar os microsserviços construídos

Uma das vantagens em se utilizar os microsserviços é que estes possuem seus próprios ciclos de compilação de maneira independente. Para tanto, faz-se necessário adotar uma solução de recipiente leve, em plataforma aberta, para administradores e desenvolvedores, que permita executar aplicações distribuídas, além de oferecer um serviço de nuvem para o compartilhamento de aplicativos e automação dos fluxos de trabalho, proporcionando isolamento de recursos necessários, como por exemplo CPU, memória e rede.

O *Docker* tecnicamente é uma plataforma aberta para se implantar e executar aplicações distribuídas, permitindo que elas sejam construídas a partir de componentes, eliminando o atrito entre o ambiente de desenvolvimento e o controle de qualidade e produção definido em seu aspecto comercial. Com o *Docker* também é possível fornecer um ambiente padronizado para a equipe de desenvolvimento, além de facilitar a implantação e execução de aplicativos em qualquer infraestrutura.

Figura 14 - Representação da tecnologia de Máquina Virtual



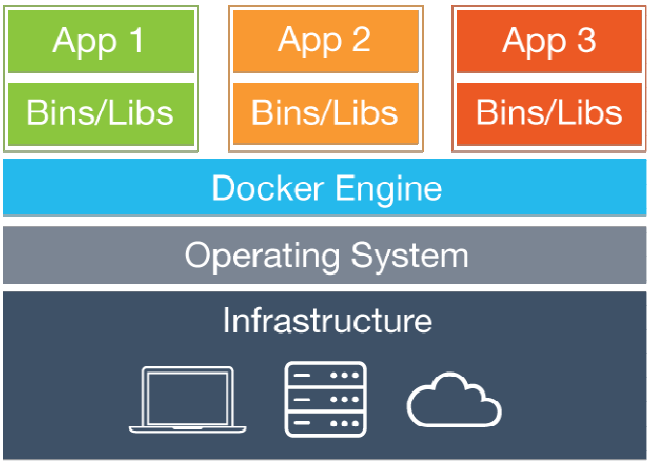
Fonte: Desmembramento dos recursos de uma máquina virtual tradicional[5]

Como mostra a Figura 14, cada aplicação virtualizada não contém apenas seu aplicativo, uma vez que na virtualização se deve incluir também os arquivos binários, as bibliotecas necessárias, além de um sistema operacional completo em operação para permitir que uma determinada aplicação se mantenha em execução como expectado, o que pode necessitar um espaço em média de 10 *Gigabytes* (GB) de armazenamento em disco.

[5] http://www.docker.com/whatisdocker/

Na Figura 15, a seguir, o recipiente provido pelo *Docker* compreende apenas o aplicativo e suas dependências, sendo que a aplicação é executada por um processo encadeado e isolado no espaço do usuário do sistema operacional em operação, compartilhando seu núcleo com os outros recipientes, aumentando a portabilidade e a eficiência em virtude particularmente da alocação e do isolamento de recursos.

Figura 15 - Representação da tecnologia Docker



Fonte: Divisão em camadas de uma tecnologia de contêiner[6]

4.7 Realizar um estudo de caso

O estudo de caso consistiu em utilizar o método proposto neste trabalho de conclusão de curso para desenvolver um sistema para auxiliar o tratamento médico com o uso de fototerapia. Este sistema é composto por um conjunto de serviços independentes, além de possibilitar acesso via dispositivos móveis.

A partir da conclusão desse estudo de caso foi possível verificar a conformidade e a adaptabilidade, bem como o refinamento do método ora proposto, tendo como base os requisitos elicitados para o desenvolvimento das aplicações e, ao final, sendo destacadas as forças e fraquezas do método no cenário e contexto em que o mesmo foi aplicado.

---

[6] http://www.docker.com/whatisdocker/

5 MÉTODO DE DESENVOLVIMENTO COM MICROSSERVIÇOS

Nesse contexto, este trabalho propõe a definição de um método para o desenvolvimento de *software* baseado em microsserviços. Este método direciona a construção de serviços, pois fornece um conjunto de nove passos bem definidos e utiliza os principais conceitos para desenvolvimento de aplicações distribuídas, utilizando alta tecnologia, computação em nuvem e a habilidade de escala horizontal e vertical, tendo seus recursos computacionais garantidos, isolados e seguros a fim de evitar a sobrecarga das máquinas físicas provenientes do alto número de acessos simultâneos dos multiusuários. Com a utilização do método proposto pretende-se facilitar o processo de desenvolvimento de sistemas de *software* mais complexos, confiáveis, modulares, com maior qualidade e voltados à grande escala.

A proposta compreende também, caso necessário, utilizar componentes de código aberto que atendam às necessidades do negócio e auxiliem na construção de um *software* baseado em microsserviços, evitando custos não planejados com licenciamento ou propriedade intelectual de *software*.

5.1 Elicitação de requisitos

Como primeira etapa deste conjunto de passos, a elicitação de requisitos é uma tarefa primordial para que se identifique as regras de negócio em que o sistema se apoia e uma maneira de simplificar e validar o entendimento das funcionalidades deste sistema. Como em todo processo de engenharia de *software* é desejável conhecimento sobre o domínio da aplicação, juntamente com o entendimento específico do problema a ser resolvido e do negócio, levando em consideração as necessidades do cliente e as limitações das tecnologias.

Pode-se utilizar desde as principais técnicas tradicionais, como:

- Coleta de dados
- Entrevistas
- Questionários
- Reuniões

Técnicas colaborativas, como:

- *Brainstorms*
- Prototipação

Técnicas cognitivas, como:

- Análise de tarefas
- Técnicas de aquisição de conhecimento

E também, abordagens contextuais, como:

- Etnografia
- Análise de discursos

Ambas técnicas citadas estão disponíveis na literatura, cada uma contendo seu próprio conjunto de vantagens e desvantagens em sua aplicação prática[7].

Acompanhando a abordagem da sessão 2.1.1 “Princípios dos Microsserviços”, os serviços devem ser projetados para serem leves, coesos e possuírem responsabilidade única, sendo assim, utilizar uma metodologia de desenvolvimento ágil, como o *Scrum* apoiado em estórias de usuário, pode auxiliar o planejamento e a gestão do projeto de *software*, pois os ciclos tipicamente mensais e as funcionalidades a serem implementadas mantidas no *Product Backlog* permitem ciclos de iterações ideais aderentes ao modelo de desenvolvimento de *software* baseado em microsserviços proposto.

5.1.1 Abordagem de desenvolvimento ágil de software com microsserviços

Uma abordagem para o desenvolvimento baseado em microsserviços é utilizar o modelo de desenvolvimento ágil, com ele é possível manter entregas contínuas do produto, separando, caso necessário, o desenvolvimento dos microsserviços entre os times responsáveis. Pode-se granular os serviços a partir de um conceito inicial e dar início aos processos que antecedem ou sucedem a própria fase de desenvolvimento, e consequentemente manter as entregas em paralelo.

Figura 16 - Quadro das fases de desenvolvimento ágil com microsserviços

Fonte: Elaborado pelo autor.

[7] http://martinfowler.com/articles/microservice-testing/

Na Figura 16, são exibidas as fases de desenvolvimento adotadas para esta proposta e como podem ser mantidos os *status* de uma determinada funcionalidade ao longo o ciclo de vida do processo de desenvolvimento dos microsserviços.

5.2 Definir funcionalidades

Esta segunda etapa, apoiada nos requisitos funcionais, não funcionais e de qualidade, consiste na definição das funções (funcionalidades) do sistema, ainda seguindo os conceitos de projeto simples, microsserviços leves e independentes, ou seja, ao definir as funcionalidades um serviço deve possuir zero dependências e estar limitado a um único domínio.

Deve ser realizada a decomposição do produto de *software* em microsserviços com base nas responsabilidades e interações do sistema, em que se identifica funcionalidades para encontrar contextos limitados. Deve-se encontrar responsabilidades específicas impostas por um limite explícito para definir as funcionalidades que compreenderão o sistema em sua totalidade.

A alternativa para definição de funcionalidades em um novo projeto é modelar serviços para microsserviços baseando-se no produto mínimo viável, em que se inicia com um pequeno conjunto de funcionalidades com foco no núcleo ou core do sistema, atentando-se à arquitetura, qualidade do código e se possível entrega contínua. A estratégia de particionamento de funcionalidades em verbos, como abordado no trabalho relacionado 3.1 "*On Micro-services Architecture*", auxilia na limitação de contextos, como por exemplo:

- Contexto do website:
    - Assinar newsletter.
    - Efetuar pagamento.
    - Fornecer informação de entrega.
    - Adicionar artigo ao carrinho de compras.
- Contexto da manipulação de pedidos:
    - Enviar e-mail de confirmação.
    - Atualizar estoque.
    - Escolher produto em armazém.
- Contexto do relacionamento com o cliente:
    - Mapear interesse do cliente.

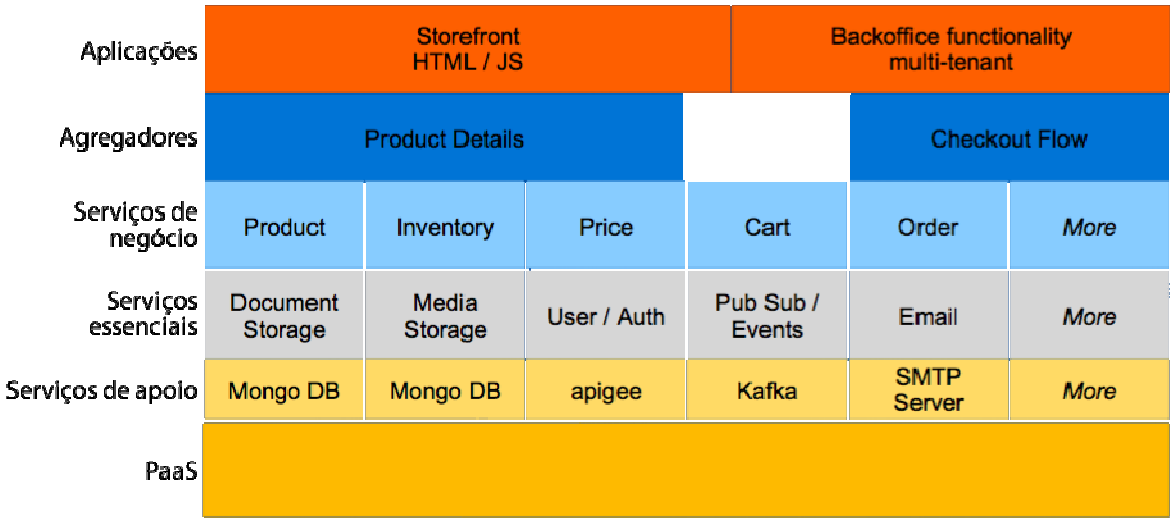
- Personalizar recomendações.

5.3 Definir relacionamento entre as funcionalidades

Neste terceiro passo, um fator importante a ser considerado após a definição e formalização das funcionalidades da etapa anterior, é utilizar o conceito de agregador para unificar as chamadas aos serviços que compõem o fluxo de relacionamento entre as funcionalidades.

Como mostra a Figura 17, a seguir, ao definir o relacionamento entre as funcionalidades ou responsabilidades, o conceito de agregador otimiza a API para a aplicação da camada superior (Aplicações) tornando seu comportamento mais consistente e principalmente promove o isolamento das funcionalidades dos serviços na camada de Serviços de negócio por meio da inversão de controle dos microsserviços para a camada de Agregadores.

Figura 17 - Exemplo de arquitetura e definição de responsabilidades por camadas



Fonte: Adaptado de hybris-as-a-service, A Microservices Architecture in Action[8]

Ainda na Figura 17, os Serviços de apoio, divididos em uma camada, são quaisquer serviços consumidos via rede como parte da operação normal ou fluxo de operação de um microsserviço essencial ou de negócio.

Sendo assim, nessa proposta, clientes na camada de Aplicações não se comunicariam diretamente com os microsserviços da camada Serviços de negócio, simplificando o número de requisições para atender a uma regra ou fluxo de negócio e

[8] Microxchg conf: http://microxchg.io/2015/slides/02_05_microxchg_andrea.pdf

reduzindo o tratamento de erros e as lógicas de negócio da responsabilidade do cliente na camada de Aplicações.

5.4 Definir serviços e interfaces fundamentado pelas funcionalidades

Ao definir serviços e interfaces para os microsserviços uma das vantagens notáveis desta arquitetura é a sua natureza granular e, com isso, o fato de se possuir várias opções a respeito de como se resolver problemas. A fim de reduzir a repetição de código entre serviços distintos que utilizam modelos semelhantes, pode-se adotar as técnicas elencadas abaixo, desde que elas não gerem dependências entre os serviços, e possam ser reaproveitadas durante a fase de desenvolvimento, como:

- Bibliotecas compartilhadas.
- Módulos.

Neste momento, com o intuito de reduzir a transmissão de dados entre cliente/servidor e minimizar o número de solicitações aos recursos disponibilizados pelos microsserviços, deve-se especificar como um cliente solicita que os recursos sejam acessados ou modificados e como o servidor deve responder a essas solicitações. Para definir serviços e interfaces mais complexas pode-se utilizar UML, em especial, os diagramas de sequência, com o intuito de representar visualmente o comportamento das funcionalidades que envolvem várias classes e chamadas a métodos distintos, tornando complexo determinar o fluxo e o comportamento ao longo do ciclo de vida do microsserviço.

Nas subseções posteriores abordam-se as responsabilidades dos clientes e servidores, bem como indica-se uma padronização para códigos de resposta e retorno frequentemente utilizados.

5.4.1 Comunicação e negociação de conteúdo entre clientes e serviços

Para o planejamento de comunicação e construção da API deve-se manter uma semântica intuitiva, sendo que os termos adotados precisam ser concretos e de fácil assimilação ao prover acesso às funcionalidades projetadas para os microsserviços. Adotar as especificações da json:api (2015) para o formato das solicitações e respostas e seguir as convenções compartilhadas permite se concentrar na comunicação dos serviços, aumentando a produtividade na fase de desenvolvimento.

O cliente deve solicitar e enviar toda informação via requisição, obrigatoriamente contendo o cabeçalho **Content-Type: application/vnd.api+json** e sem nenhum tipo de parâmetro adicional (JSON:API, 2015).

O servidor deve enviar toda informação de resposta via documento, obrigatoriamente contendo o cabeçalho **Content-Type: application/vnd.api+json** e sem nenhum tipo de parâmetro adicional (JSON:API, 2015). Neste contexto, documentos são quaisquer estruturas definidas em JSON, ou seja, *JavaScript Object Notation*, um formato leve de intercâmbio de dados.

5.4.2 Códigos de resposta e retorno

Ao planejar o desenvolvimento da API, deve-se padronizar os códigos de resposta e retorno, pois eles indicam ao cliente um *feedback* sobre a requisição solicitada.

- 200: Sucesso.
- 400: URL Inválida.
- 403: Recurso inválido.
- 404: Recurso não encontrado.
- 412: Falha na requisição.
- 500: Erro interno de servidor.
- 503: Limite de uso atingido.

Assim, define-se uma especificação de como os clientes devem solicitar os recursos e como o servidor deve responder a esses pedidos. Com essa definição, os clientes podem padronizar tomadas de decisão sobre alguma falha inesperada ao acessar recursos ou caso parte de um fluxo seja interrompido baseando-se nos códigos de resposta definidos. Se necessário, pode-se utilizar customizar mais códigos de resposta, desde que se adequem ao fluxo de execução.

5.5 Escolher tecnologia para implementação de microsserviços

Neste quinto passo, deve-se optar pela tecnologia de construção dos microsserviços que mais atenda às necessidades do sistema, obedecendo ao princípio da responsabilidade única, discutido na sessão 2.1 “Microsserviços”, segregação de interfaces e inversão de controle e dependência. Nos próximos tópicos são apresentadas as principais tecnologias utilizadas para

desenvolvimento com microsserviços, seguido de ferramentas de análise de *logs*, monitoramento e localização e descoberta, para complementar os conceitos propostos neste método.

5.5.1 Principais tecnologias de microsserviços

Nesta sessão são listadas as principais tecnologias de microsserviços consolidadas e em produção no mercado, elas podem ou não ser equivalentes ou complementares, cada uma com sua particularidade e adaptabilidade ao contexto de aplicação.

Spring Boot[9]: é um *framework* não intrusivo, que auxilia a produtividade e o desenvolvimento de aplicações Java independentes. Ele foi desenvolvido baseado nos padrões de projeto de injeção de dependência e inversão de controle, possuindo uma arquitetura baseada em interfaces que oferecem mecanismos de controle de transações e segurança, além de facilitar o desenvolvimento com testes unitários. Por fornecer uma ferramenta de linha de comando que executa scripts, o *Spring Boot* proporciona uma experiência de desenvolvimento rápido e acessível, fornecendo funcionalidades comuns a uma grande categoria de projetos, como: servidores embutidos, segurança, métricas e configurações exteriorizadas programáticas sem exigência de configuração via arquivos XML.

Dropwizard[10]: é um *framework* Java para o desenvolvimento baseado em operações amigáveis ou *ops-fiendly*, de alto desempenho e com suporte a serviços *web RESTfull*. Ele reúne bibliotecas maduras e estáveis do ecossistema Java em um pacote simplificado e leve, permitindo aos desenvolvedores se concentrar no desenvolvimento. O *Dropwizard* possui configuração sofisticada, com métricas de aplicação, *logging* e ferramentas operacionais que auxiliam a desenvolver serviços de qualidade com menor tempo de produção.

Play 2[11]: é um *framework* de alta velocidade para Java e Scala[12], ele facilita a construção de aplicações por possuir uma arquitetura *leve*, sem estado e voltada à *web*. Desenvolvido em Akka[13], um toolkit de desenvolvimento concorrente, distribuído e resiliente baseado em JVM, ou Máquina Virtual Java, o framework proporciona um consumo mínimo de recursos previsíveis (CPU, memória, *threads*) para aplicações altamente escaláveis.

---

[9] http://projects.spring.io/spring-boot/
[10] http://www.dropwizard.io/0.9.2/docs/
[11] https://www.playframework.com
[12] http://www.scala-lang.org
[13] http://akka.io

Hystrix[14]: é uma biblioteca de tolerância a falhas desenvolvida pela Netflix OSS, ela foi projetada para isolar os pontos de acesso a sistemas remotos, serviços e bibliotecas de terceiros. A biblioteca ainda possui suporte a tratamento de falhas e possibilita resiliência em sistemas distribuídos complexos, onde falhas geralmente são inevitáveis.

Netflix OSS[15]: O *Netflix Open Source Software Center* é um repositório de sistemas de código fonte aberto, aproveitados, desenvolvidos, utilizados e mantidos pela Netflix, nele são disponibilizadas ferramentas para gerenciamento e controle de dados (persistentes e semi-persistentes) e ainda ferramentas para análise que auxiliam a tomada de decisão e a melhoria contínua dos serviços fornecidos ao cliente final. Os sistemas foram planejados para operar os serviços com máximo desempenho, segurança e confiabilidade, concentrando-se na construção contínua e integração das aplicações.

5.5.1.1 Análise de logs

Kibana[16]: é uma ferramenta *open source*, multiplataforma, *web based*, projetada para análise, pesquisa e visualização de *logs* utilizando o *Elasticsearch*[17] um *engine*, ou motor de busca escalável. A ferramenta foi desenvolvida para ser rápida de configurar e começar a usar, sendo poderosa e flexível assim como o *Elasticsearch*. Ela também pode ser utilizada como um *plugin* para visualização de dados abertos, fornecendo recursos de visualização de conteúdo e indexando grandes volumes de dados.

Logstash[18]: é uma ferramenta *open source* para gerenciar eventos e *logs*, ela pode ser utilizada para coletar, analisar e armazenar *logs* para uso posterior. Caso o utilize com o *Elasticsearch* ele é facilmente integrado para visualização e análise com o *Kibana*.

Loggly[19]: é um serviço proprietário baseado em nuvem, para mineração de grandes volumes de dados de *logs* em tempo real que auxilia ao produzir e analisar a qualidade de código de sistemas e melhorar a experiência do usuário.

---

[14] https://github.com/Netflix/Hystrix
[15] http://netflix.github.io
[16] https://github.com/elastic/kibana
[17] https://www.elastic.co
[18] https://www.elastic.co/products/logstash
[19] https://www.loggly.com/

5.5.1.2 Tecnologias de monitoramento

Pagerduty[20]: é um sistema proprietário de gestão de incidentes e monitoramento de tecnologias da informação, ele fornece alertas, políticas de escalonamento e monitoramento para aumentar a disponibilidade de aplicativos, servidores, *sites* e banco de dados.

Servo[21]: é um projeto da Netflix OSS que oferece uma biblioteca para monitoramento de aplicações escritas em Java. Ele fornece uma interface padrão de monitoramento que pode ser consultada por diversas ferramentas existentes e auxilia ao expor e publicar métricas sem ter que escrever código do zero.

Atlas[22]: também é um projeto da Netflix OSS que fornece uma infraestrutura para o monitoramento e gerenciamento de dados em série temporal dimensional.

Graphite[23]: é uma ferramenta *open source*, *web based* para acompanhar e representar graficamente o desempenho de sistemas, ela coleta, armazena e exibe dados de série temporal em tempo real e renderiza os gráficos sob demanda para o usuário final.

Zipkin[24]: é um sistema de rastreamento distribuído *open source* desenvolvido pelo *Twitter*[25], ele auxilia a reunir dados necessários para solucionar problemas de latência em arquiteturas baseadas em microsserviços.

5.5.1.3 Localização e descoberta de serviços

Zookeeper[26]: é um serviço centralizado criado e mantido pela *Apache Software Fountation*[27] para persistir informações de configuração, proporcionando sincronia distribuída e prestação de serviços de *software* em grupo. Com ele é possível desenvolver e manter um servidor de código aberto que permite coordenação distribuída altamente confiável.

Curator[28]: também mantido pela *Apache*, é um conjunto de bibliotecas que tornam o uso do *Zookeeper* mais simples. O *Curator* é um substituto para o cliente do *Zookeeper* que

[20] https://www.pagerduty.com/
[21] https://github.com/Netflix/Servo
[22] https://github.com/Netflix/Atlas
[23] http://graphite.wikidot.com
[24] https://github.com/openzipkin/zipkin
[25] https://twitter.com
[26] https://zookeeper.apache.org
[27] http://apache.org
[28] http://curator.apache.org

se aplica a tarefas de baixo nível com utilitários que viabilizam funcionalidades a fim de simplificar a complexidade do gerenciamento de operações em *cluster*.

Eureka[29]: outro projeto criado e mantido pela Netflix OSS, é um registrador e localizador de serviços para balanceamento de carga resiliente *mid-tier*, ou, disposto entre intermédio do cliente e servidor, com suporte a *failover*, ou seja, projetado com suporte para proteção de falhas disponibilizando recursos em espera a fim de assumir o controle quando ocorrer interrupções no sistema principal.

5.5.1.4 Gerenciamento de configurações

Archaius[30]: outro projeto da Netflix OSS, ele fornece um conjunto de APIs para gerenciamento de arquivos de configurações que auxilia a criar e customizar propriedades dinâmicas em aplicações escritas em Java podendo ser customizadas em tempo de execução.

5.6 Definir alocação dos microsserviços

Na sexta etapa deste método, projeta-se a alocação dos microsserviços em ambientes individuais, com mecanismos leves, que simplifiquem a escalabilidade, portabilidade e a replicação. Como evidenciado na sessão 4.6 “Implantar os microsserviços construídos”, a alocação dos microsserviços em um ambiente de contêiner é uma alternativa inicial ideal, partindo da necessidade da gestão de isolamento, controle do ciclo de vida, velocidade de publicação e disponibilização, tarefas relativamente comuns da administração de sistemas. Tecnologias de contêiner, como o *Docker*, facilitam o empacotamento, gerenciamento, execução e atualização da aplicação, visto que, ao incluir as dependências de um microsserviço a um contêiner, permite portá-lo para plataformas distintas, tornando a utilização dos recursos mais eficiente em comparação com as plataformas tradicionais de virtualização, proporcionando assim construir um ambiente com recursos rigorosamente planejados e controlados.

Utilizar a solução técnica de contêiner exige uma avaliação das motivações para o desenvolvimento baseado em microsserviços e principalmente a propensão à inovação futura. O uso desta abordagem não necessariamente é obrigatório e depende dos princípios

[29] https://github.com/Netflix/eureka
[30] https://github.com/Netflix/Archaius

arquitetônicos adotados, das políticas de segurança, do controle de acesso e da complexidade do sistema; deve-se buscar utilizar tecnologias que agreguem valor ao sistema em sua completude e que favoreçam sua evolução.

5.7 Testar microsserviços

A sétima etapa recomenda atividades de testes durante e após o desenvolvimento baseado em microsserviços proposto, visto que falhas podem ser motivadas por razões variadas. Segundo Myers et al. (2011), não se pode garantir que todo *software* funcione corretamente e sem a presença de erros, portanto, utilizar no mínimo a abordagem de testes de unidade ou TDD durante a construção dos microsserviços é uma alternativa altamente recomendável e desempenha um papel importante, sendo que ela acomete classes e componentes isolados para cada microsserviço projetado, no entanto, utilizar somente a abordagem de testes unitários não fornece garantias sobre o comportamento do sistema.

Deve-se escolher dentre os tipos de testes de *software* as estratégias que preferencialmente adicionem valor ao sistema, seguindo o objetivo proposto pelo teste e gerenciando a complexidade adicional pela sua adoção. A seguir, são elencados os 13 principais tipos de teste de *software* que podem ser aplicados em microsserviços:

1. Testes de caixa branca e caixa preta.
2. Testes de configuração.
3. Testes funcionais.
4. Testes de instalação.
5. Testes de integridade.
6. Testes de integração.
7. Testes de manutenção.
8. Testes de performance.
    - Testes de carga.
    - Testes de stress.
9. Testes de regressão.
10. Testes de segurança.
11. Testes de unidade.
12. Testes de usabilidade.
13. Testes de volume.

Mais detalhes e aplicações práticas sobre cada tipo de testes de *software* citados acima estão amplamente disponíveis na literatura[31] [32]. A seguir são elencadas algumas tecnologias e ferramentas para desenvolvimento orientado à testes.

JUnit[33]: é um *framework open source* para criação de testes automatizados utilizando Java como linguagem de programação.

Cucumber[34]: é uma ferramenta de *software* para automatizar testes de aceitação escritos no estilo *Behavior-Driven Development* ou BDD.

Selenium[35]: é um *framework* de testes de *software* portátil para aplicações *web*.

5.7.1 Entrega contínua

Dependendo da abordagem de testes empregada e da necessidade de adições e modificações frequentes de funcionalidades, neste contexto de desenvolvimento baseado em microsserviços, optar pela entrega contínua pode simplificar a liberação e refatoração de funcionalidades para o cliente de maneira segura, sem a necessidade de haver intervenção manual dos desenvolvedores, atualização de ambientes e verificação/validação da integridade dos testes unitários. Apesar dessa técnica prover uma complexidade adicional aos projetos de *software* e possivelmente exigir profissionais capacitados, pode ser uma alternativa crucial baseando-se na necessidade de liberação e nas políticas de entrega de *software* empregadas.

5.8 Implementação

A implementação de um microsserviço é a etapa final deste método proposto, tendo em vista uma abordagem focada no desenvolvimento; uma das vantagens dessa metodologia é poder escolher a tecnologia que mais se adequa para solucionar um determinado problema, não impondo à mesma os demais serviços. É importante que a tecnologia adotada suporte as definições de comunicação e negociação de conteúdo através da rede, sem afetar a estruturação da aplicação e se adaptando ao ambiente de negócio dinâmico.

---

[31] http://martinfowler.com/articles/microservice-testing/
[32] https://www.oreilly.com/learning/building-microservices-testing
[33] http://junit.org/
[34] https://cucumber.io
[35] http://docs.seleniumhq.org

5.8.1 Implementação dos microsserviços

Os microsserviços construídos para o estudo de caso, aprofundado no próximo capítulo, foram desenvolvidos utilizando o *framework Spring Boot*, em que cada serviço possui seu próprio mecanismo de persistência e uma estrutura genérica pronta para ser distribuída. A utilização do *Spring Boot* com o *Maven*[36] neste contexto simplifica a estruturação da arquitetura, a construção da aplicação e viabiliza sua execução gerando um único arquivo JAR executável e portável do microsserviço (com todas dependências necessárias), que pode ser facilmente inicializado via linha de comando ou replicado para ambientes distintos. A estrutura desse *framework* também permite a criação de testes unitários automatizados a partir da inicialização do projeto, sendo assim, é possível desencadear ações para testar funcionalidades e comportamentos. O *Spring Boot* também conta com uma série de *endpoints* que permite monitorar e interagir com os microsserviços com base em informações básicas de memória, lista de configurações, ambiente de execução, métricas, mapeamento de requisições e informações de trace para pedidos de requisições. O *framework* se mostrou uma ótima alternativa para entusiastas da abordagem de microsserviços e maduro no mesmo âmbito para suportar o desenvolvimento de aplicações robustas e complexas.

Complementando a proposta do capítulo quinto, este método foi projetado para minimizar tanto o número de solicitações, quanto a quantidade de dados transmitidos entre microsserviços. Essa eficiência deve ser priorizada ao longo do tempo, desde que se evite comprometer a legibilidade e a flexibilidade dos microsserviços.

5.9 Monitoramento dos microsserviços

Para esta etapa semifinal, com base nas ferramentas citadas na sessão 5.5.1.2 “Tecnologias de monitoramento”, deve-se monitorar a utilização e acompanhar a execução tanto quanto o fluxo de dados dos microsserviços essenciais para verificar a performance e a necessidade de escala. Essa etapa apresenta uma série de desafios, principalmente para aplicações escaláveis que adotam tecnologia de contêiner, e pode ter a complexidade de adoção influenciada pela utilização de protocolos de troca e enfileiramento de mensagens entre diferentes serviços.

[36] https://maven.apache.org

Uma ferramenta indicada para este contexto é o *spigo*[37], ou seja, *Simulate Protocol Interactions in Go using nanoservice actors*, ele é um protocolo em desenvolvimento de simulação de interações, que também auxilia a visualização tanto da rede de interações quanto do fluxo de requisições, gerando um mapa visual em forma de grafo baseado no contexto das trocas de mensagens, o que pode simplificar o entendimento da arquitetura e da comunicação entre serviços.

Para monitorar microsserviços de maneira efetiva, deve-se lidar com mudanças frequentes, uma tarefa corriqueira em ambientes em que se adota o desenvolvimento baseado em microsserviços podendo vir a ser um dos principais desafios para manter a adaptabilidade às ferramentas de monitoramento.

[37] https://github.com/adrianco/spigo

6 ESTUDO DE CASO

Este estudo de caso consiste em implementar uma aplicação funcional básica no contexto de auxílio ao tratamento médico com o uso de fototerapia, utilizando o método de desenvolvimento de *software* baseado em microsserviços proposto neste projeto de pesquisa e trilhando os requisitos funcionais elementares necessários para a construção do sistema.

Enquanto sistemas monolíticos possuem suas funcionalidades e regras de negócio desenvolvidas e implementadas sob uma única base de código ou unidade, uma aplicação baseada em microsserviços desagrega as responsabilidades em serviços leves, coesos, independentes e especializados. Dessa maneira, há um sistema principal, neste estudo definido como a interface gráfica com o usuário, e outros sistemas baseados em microsserviços que gravitam em torno deste sistema principal.

Sendo assim, quando um evento é disparado em virtude de uma interação do usuário através do sistema principal, os diferentes microsserviços essenciais são notificados. Como definido na sessão 2.3 “REST como modelo arquitetural”, essas notificações são encapsuladas nas requisições HTTP e os microsserviços responsáveis operam em conjunto, compartilhando dados em mensagens no formato JSON. Cada um dos microsserviços decide o que deve ser realizado com base nos dados recebidos e conforme a necessidade de negócio do sistema como um todo. Essa abordagem característica contribui para o desacoplamento dos microsserviços, pois a independência entre eles reduz a existência de um único ponto de falha e, caso ocorra alguma intermitência em algum dos serviços ou entre os canais de comunicação, a disponibilidade do sistema principal não é comprometida.

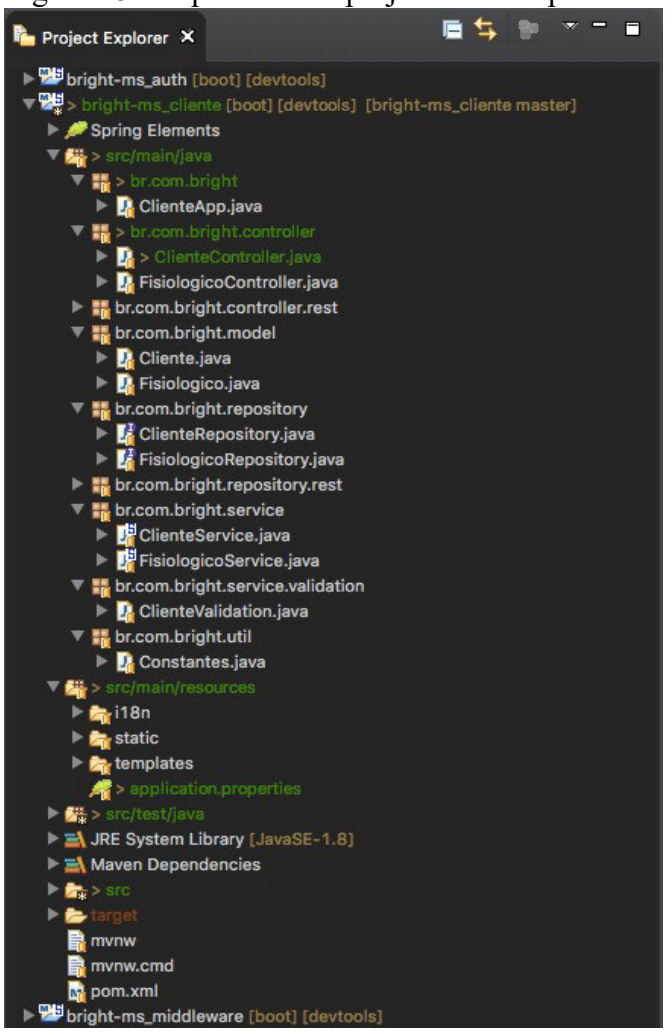
A abordagem supracitada também facilita a adoção de tecnologias distintas, os seja, pode-se dispor de serviços do próprio sistema principal e inclusive de mecanismos de persistência de dados utilizados pelos serviços sendo mantidos em tecnologias distintas.

Um fator observado durante o desenvolvimento com microsserviços foi a intersecção de código entre os serviços independentes. Classes que desempenham papel de modelo frequentemente deveriam ser replicadas entre os serviços para tratamento dos objetos comuns, causando duplicidade de informação. Para atacar esse problema foi adotada a abordagem de criação de bibliotecas compartilhadas entre projetos.

Para o cenário específico deste estudo de caso foram desenvolvidas cinco funcionalidades de negócio que englobam os microsserviços de *middleware* (bright-ms_middleware), autenticação (bright-ms_auth) e gerenciamento de clientes (bright-ms_cliente) em conjunto, onde os microsserviços fazem parte do pacote de domínio

completamente expressado “br.com.bright”, com o intuito de simplificar a referência à aplicação que representa o serviço e ambos possuem como padrão a arquitetura planejada e apresentada na Figura 18 a seguir.

Figura 18 - Explorador de projetos do Eclipse Java EE IDE 4.5.1



Fonte: Elaborado pelo autor

Cada microsserviço foi desenvolvido seguindo a estruturação de código modelo-visão-controlador ou MVC, um padrão arquitetural que facilita a separação entre interação do usuário e a representação da informação no sistema.

O microsserviço de *middleware* age como um registrador básico de recursos, ele encontra, conhece e verifica o *status* de execução a partir das informações de saúde dos microsserviços disponíveis e indisponíveis que compõem a aplicação em sua totalidade. Cada microsserviço no contexto deste estudo de caso possui uma série de mapeamentos *default* para seus pontos de extremidade, contendo informações de monitoramento diversas convencionadas pela comunidade. Estes mapeamentos são nomeados de *endpoints* atuadores, e facilitam tanto

a interação quanto o próprio monitoramento da aplicação. Por exemplo, informações de saúde, como quando o serviço é requisitado através de um canal de comunicação seguro ou não para o microsserviço *bright-ms*_cliente são mapeados para a URL /heath, como mostra a Tabela 2, que também expõe uma série de mapeamentos habilitados para este mesmo serviço.

Tabela 2 – Mapeamento padrão de *endpoint* do microsserviço bright-ms_cliente

ID	Descrição
actuator	*Provides a hypermedia-based "discovery page" for the other endpoints*
autoconfig	*Displays an auto-configuration report showing all auto-configuration candidates*
beans	*Displays a complete list of all the Spring beans in your application*
dump	*Performs a thread dump*
env	*Exposes properties from Spring's Configurable Environment*
health	*Shows application health information*
info	*Displays arbitrary application info*
metrics	*Shows 'metrics' information for the current application*
mappings	*Displays a collated list of all Request Mapping paths*
trace	*Displays trace information (by default the last few HTTP requests)*

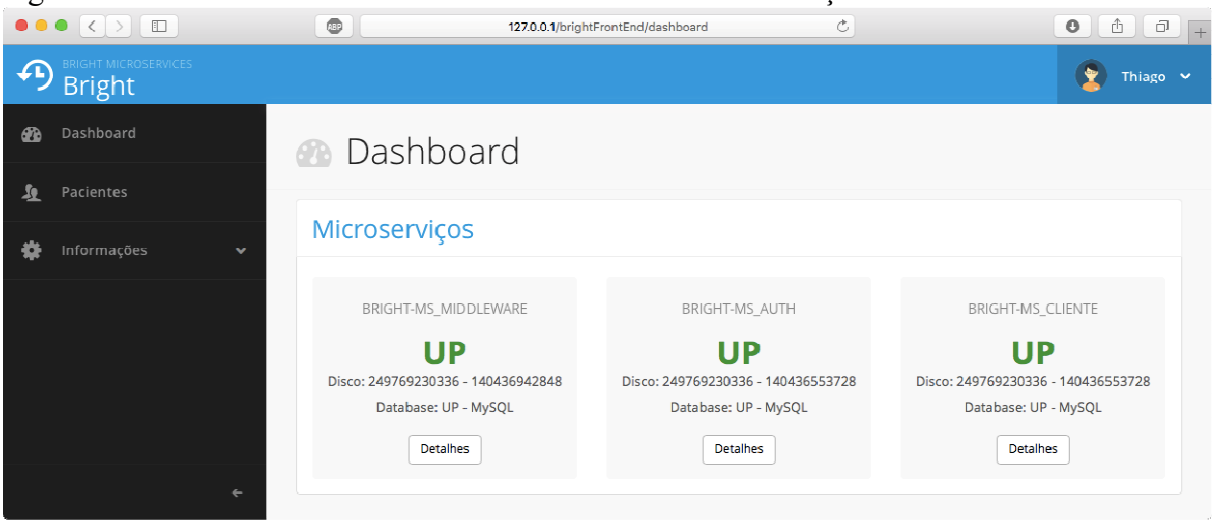
Fonte: Elaborado pelo autor

O microsserviço de autenticação estabelece e confirma a autenticidade do usuário, processa requisições de *login*, *logout*, cria e verifica o *token* de autenticidade e checa a procedência dos clientes, com a finalidade de preservar a sessão entre os contextos distintos de cada serviço.

O microsserviço de cliente persiste e gerencia os dados pessoais dos clientes, permitindo novos cadastros, visualização, alterações e exclusão dos registros existentes e também disponibiliza a informação referente aos dados pessoais e fisiológicos dos pacientes aos demais microsserviços.

Na Figura 19 a seguir, é apresentado o *dashboard*, uma interface de administração que exige autenticação, desenvolvida com o intuito de exibir informações sintetizadas sobre o *status* de execução dos microsserviços individuais que compõem o sistema em sua completude.

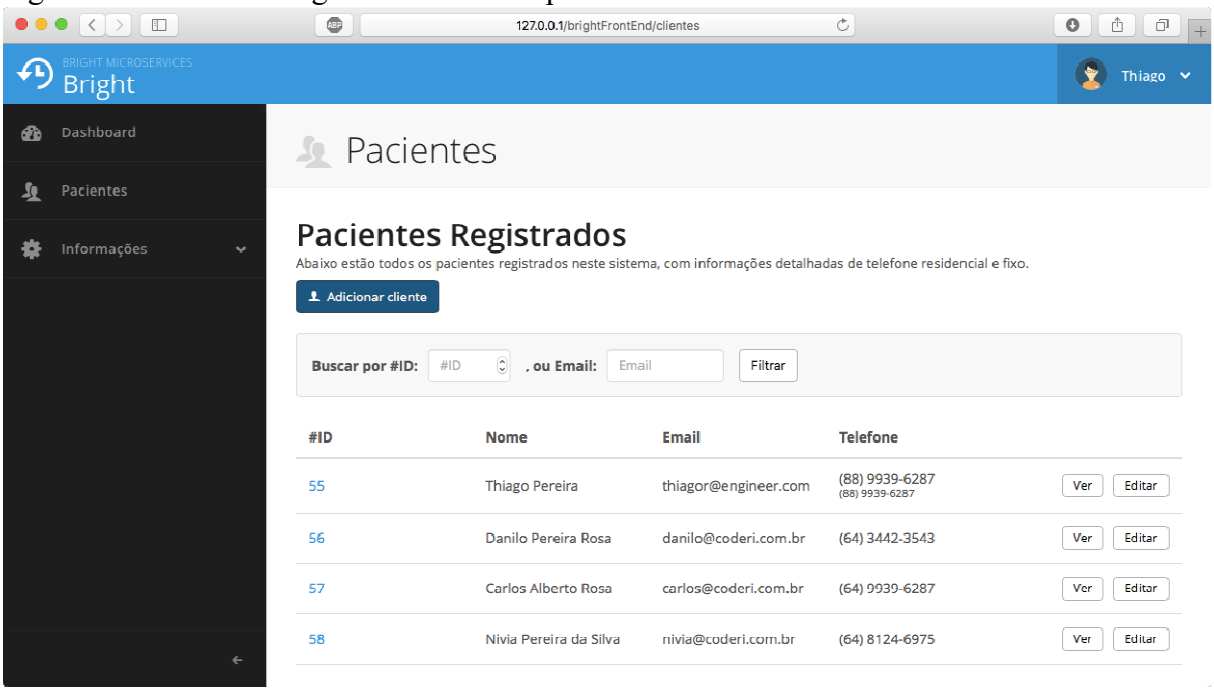
Figura 19 - *Dashboard* de monitoramento básico de microsserviços



Fonte: Elaborado pelo autor

A Figura 20 apresenta a interface de administração para gerenciamento de pacientes. Nela, após realizar autenticação, são exibidas informações resumidas de todos os clientes registrados, as operações que podem ser realizadas para os registros individuais sendo possível também realizar uma busca para facilitar a localização de um registro específico.

Figura 20 - Interface de gerenciamento de pacientes

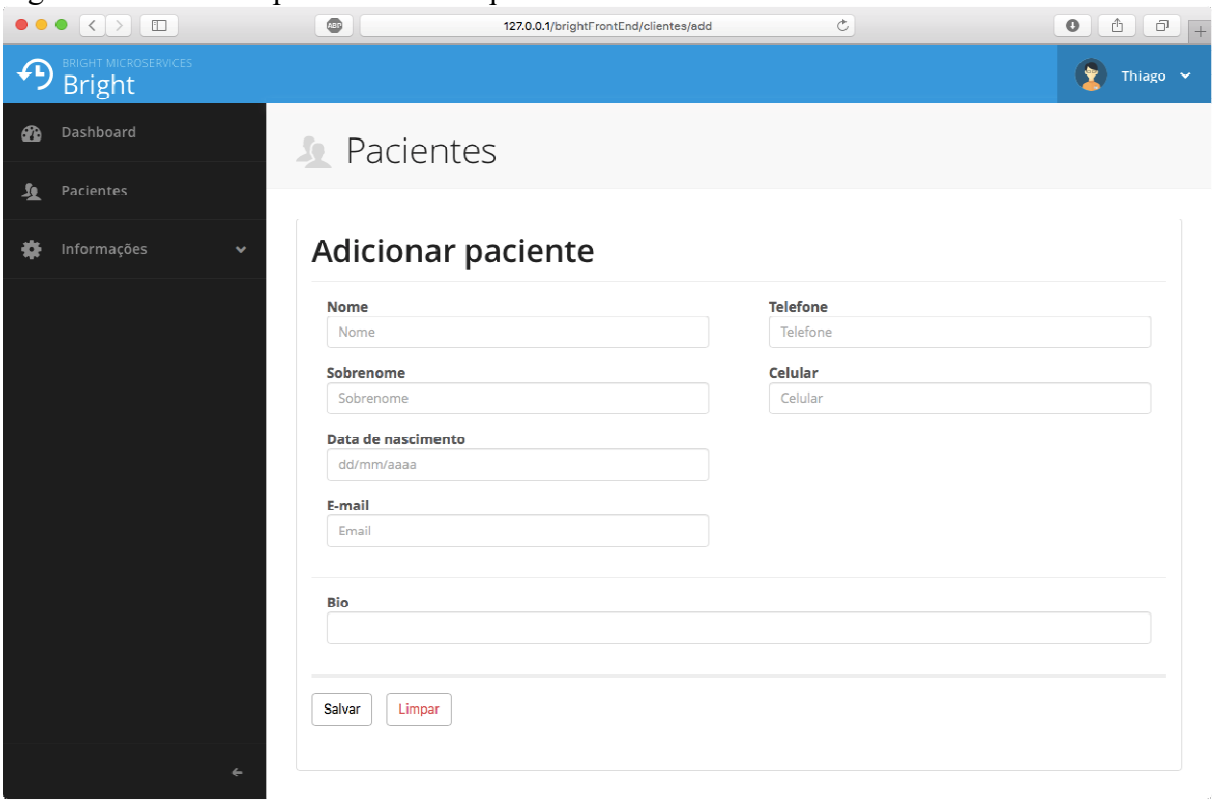


Fonte: Elaborado pelo autor

A Figura 21 apresenta mais uma interface de administração para efetuar o cadastro de pacientes, ela também exige autenticação. Nela são preenchidos os dados pessoais de uma

pessoa definida através de um formulário *web* a fim de se persistirem as informações pessoais e fisiológicas dos pacientes, neste cenário, uma responsabilidade igualmente do microsserviço bright-ms_cliente.

Figura 21 - Interface para inclusão de paciente



Fonte: Elaborado pelo autor

Seguindo os conceitos propostos e os nove passos recomendados ao longo deste projeto de pesquisa, a interface gráfica com o cliente e o *core* do sistema principal foram desenvolvidos utilizando as tecnologias PHP (*PHP: Hypertext Preprocessor*), Javascript e CSS, diferente da tecnologia Java empregada na construção dos microsserviços. A utilização destas tecnologias é opcional, e no contexto deste trabalho foram adotadas para agilizar o desenvolvimento. Cada microsserviço individual e o sistema principal foram divididos em projetos isolados com o intuito de inteligibilidade, simplificar o gerenciamento, aumentar a manutenibilidade e favorecer sua evolução. A utilização do sistema principal pelo usuário final torna transparente o *backend* do sistema em sua totalidade, já que ele não precisa conhecer, localizar, nem verificar a disponibilidade dos microsserviços que o orbitam para mantê-lo disponível em sua totalidade. Os microsserviços foram testados local e geograficamente e distribuídos com o intuito de avaliar o comportamento da interface principal, como ela lida com a divergência de latência ponto a ponto e com atrasos na comunicação, mostrou-se apta a operar

sob conexões com baixa taxa de transferência de dados, sempre exibindo um *feedback* ao usuário final em relação ao carregamento assíncrono dos conteúdos sem comprometer o carregamento das páginas ou *pageload* do sistema principal não afetando em termos de latência a experiência do usuário final.

6.1 Repositório de código do estudo de caso

O código fonte do projeto deste estudo de caso, incluindo casos de uso, microsserviços, aplicação principal e diagramas das estruturas de banco de dados com instruções SQL estão disponíveis no seguinte repositório:

- https://bitbucket.org/kamihouse/bright_microservices

7 CONCLUSÃO

O método de desenvolvimento proposto neste trabalho de conclusão de curso possui um conjunto de passos que auxilia a criação e o projeto de sistemas de *software* complexos e confiáveis baseados em microsserviços, pois leva em consideração as fases que antecedem o desenvolvimento, como elicitação e definição de requisitos, e fases pós desenvolvimento, como testes e manutenção, particularmente orientadas à serviços.

Como este método se apoia na abordagem de desenvolvimento ágil, ele considera a modularização e granulação de serviços durante as fases de planejamento e implementação. Ao se tratar de gerenciamento de projeto, ele auxilia o planejamento, a definição e a alocação de atividades entre o time de desenvolvimento, levando em consideração os pontos identificados como cruciais para a abordagem de desenvolvimento voltado à microsserviços, como a separação de interesses, a diferenciação de domínios de negócio na identificação de relacionamentos, o reconhecimento de grupos de funcionalidades que são bons candidatos a se tornar um microsserviço, e ainda, atrai a atenção para as estruturas de comunicação e interação entre serviços.

Este método em integralidade pode ser considerado demasiado complexo, dispendioso, e em termos de processo, pesado para construção de sistemas cuja tendência de crescimento e disponibilidade seja considerada constante, com menor exigência de desempenho e de baixa elasticidade. Para aplicações demasiadamente simples, esse método exige uma heterogeneidade tecnológica e como consequência, camadas adicionais são necessárias para suportar a execução do sistema como um todo. E por fim, para abordar diferente tipos e contextos de *software*, foi elencado um conjunto de tecnologias de desenvolvimento, *logging*, monitoramento, localização, descoberta e gerenciamento de microsserviços, atualmente em evidência e adotadas no cenário empresarial, a fim de orientar o leitor às tecnologias consolidadas no mercado.

Para trabalhos futuros, utilizar o protocolo AMQP ou *Advanced Message Queueing Protocol* em lugar do REST, visto que o AMQP é um padrão para troca de mensagens assíncronas interoperável, devido seu formato de mensagens e transmissão padronizados ele permite escrever bibliotecas utilizando linguagens, tecnologias distintas e principalmente executar aplicações multiplataforma, não atrelando o desenvolvimento de aplicações à arquitetura de CPU ou a sistemas operacionais específicos. O AMQP é um mecanismo que coordena o envio e recepção de mensagens em uma estrutura de fila, o que permite projetar um

padrão para o protocolos de mensageria, sendo assim, pode-se optar pela utilização de *Remote Procedure Call* ou Chamada Remota de Procedimento, encapsulada de maneira transparente pelo próprio AMQP; No contexto do estudo de caso deste TCC, a tecnologia *open source RabbitMQ*[38] especializada em efetuar a comunicação entre serviços via plataforma de mensageria seria uma forte candidata a ser adotada.

O principal beneficio dessa abordagem, mesmo integrando uma camada adicional de tecnologia é a capacidade de monitorar em tempo real a utilização, performance e a troca de mensagens entre os microsserviços, auxiliando a tomada de decisões sobre escalabilidade. Ainda como trabalhos futuros, para aplicações de baixa latência e altamente disponíveis, ou para aplicações de tempo real, a utilização da abordagem de programação reativa, com utilização de *frameworks* como o *Meteor*[39], ou o *Vert.x*[40] aplicados nas camadas de serviços e principalmente no *front-end*, podem influenciar positivamente a estrutura, o fluxo de dados, apoiar a propagação de mudanças e facilitar a escala com utilização de múltiplas instancias de componentes.

[38] https://www.rabbitmq.com
[39] https://www.meteor.com
[40] http://vertx.io

REFERÊNCIAS

BERNERS-LEE, Tim et al. **Uniform Resource Locators (URL).** In: Network Working Group, University of Minnesota, 1994. Disponível em: <http://www.ietf.org/rfc/rfc1738.txt>. Acesso em: 23 set. 2015.

Cunningham & Cunningham, Inc. **Monolithic Design**. Disponível em: <http://c2.com/cgi/wiki?MonolithicDesign>. Acesso em: 12 jan. 2016.

Docker. Disponível em: <https://www.docker.com>. Acesso em: 17 fev. 2016.

ELMANGOUSH, Asma; AL-HEZMI, Adel; MAGEDANZ, Thomas. Towards Standard M2M APIs for Cloud-based Telco Service Platforms. In: **Proceedings of International Conference on Advances in Mobile Computing & Multimedia**. ACM, 2013. p. 143.

EUROPEAN TELECOMMUNICATIONS STANDARDAS INSTITUTE. **ETSI Machine-to-Machine Communications info and drafts**. Disponível em: <http://docbox.etsi.org/M2M/Open/>. Acesso em: 17 abr. 2015.

FIELDING, Roy Thomas. **Architectural styles and the design of network-based software architectures**. 2000. Tese de Doutorado. University of California, Irvine.

FIELDING, Roy T.; TAYLOR, Richard N. Principled design of the modern Web architecture. **ACM Transactions on Internet Technology (TOIT)**, v. 2, n. 2, p. 115-150, 2002.

FOWLER, Martin. **The New Methodology**. Disponível em: <http://www.martinfowler.com/articles/newMethodology.html#N8B>. Acesso em: 2 mai. 2015.

FOWLER, Martin. **Patterns of enterprise application architecture**. Addison-Wesley Longman Publishing Co., Inc., 2002.

GARLAN, David; SHAW, Mary. **An introduction to software architecture.** 1994. Disponível em: <http://repository.cmu.edu/cgi/viewcontent.cgi?article=1720&context=compsci>. Acesso em: 20 abr.2015.

JEN, Lih-ren; LEE, Yuh-jye. Working Group. IEEE recommended practice for architectural description of software-intensive systems. In: **IEEE Architecture**. 2000. Disponível em <https://standards.ieee.org/findstds/standard/1471-2000.html>. Acesso em: 7 mai. 2015.

Json:api. Disponível em: <http://jsonapi.org>. Acesso em: 23 set. 2015.

KUNZE, J. Functional Requirements for Internet Resource Locators. Work in Progress, December, 1994.

LEWIS, James; FLOWER, Martin. **Microservices**. Disponível em: <http://martinfowler.com/articles/microservices.html>. Acesso em: 3 abr. 2015.

LEVCOVITZ, Alessandra; TERRA, Ricardo; VALENTE, Marco Tulio. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems.

MARINESCU, Dan C. **Cloud computing:** Theory and practice. 1 ed. Waltham: Newnes, 2013.

MARTIN, Robert Cecil. **Agile software development:** principles, patterns, and practices. Prentice Hall PTR, 2003.

MARTIN, Robert Cecil. **The Clean Architecture.** Disponível em: <http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Acesso em: 25 mai. 2015.

MARTIN, Robert C. The single responsibility principle. **The Principles, Patterns, and Practices of Agile Software Development**, p. 149-154, 2002.

Microservices, Docker and Containers, an Overview. Disponível em: <http://www.simplicityitself.com/articles/microservices-docker-and-containers-an-overview>. Acesso em: 25 jan. 2016.

Microxchg - The Microservices Conference in Berlin. Disponível em: <http://microxchg.io>. Acesso em: 23 jan. 2016.

MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. **The art of software testing.** John Wiley & Sons, 2011.

NAMIOT, Dmitry; SNEPS-SNEPPE, Manfred. On M2M Software. **International Journal of Open Information Technologies**, v. 2, n. 6, p. 29-36, 2014.

NAMIOT, Dmitry; SNEPS-SNEPPE, Manfred. On micro-services architecture. **International Journal of Open Information Technologies**, v. 2, n. 9, p. 24-27, 2014.

NEWMAN, Sam. **Building Microservices**. 1 ed. O'Reilly Media, Inc., 2015.

PRESSMAN, Roger S. **Engenharia de Software**. 6 ed. McGraw-Hill, 2006.

RICHARDSON, Chris. **Microservices architecture.** Disponível em: <http://microservices.io/patterns/microservices.html>. Acesso em: 20 mai. 2015.

RICHARDSON, Chris. Microservices: Decomposing applications for deployability and scalability. 2014.

ROBERTS, Wendy E. Skin type classification systems old and new. **Dermatologic clinics**, v. 27, n. 4, p. 529-533, 2009.

SCRUM. Disponível em: <http://www.desenvolvimentoagil.com.br/scrum>. Acesso em: 22 jan. 2016.

SOUSA, Flávio RC; MOREIRA, Leonardo O.; MACHADO, Javam C. Computação em

nuvem: Conceitos, tecnologias, aplicações e desafios. **II Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI)**, p. 150-175, 2009.

TEAM, CMMI Product. CMMI for Development, version 1.2. 2006.

Testing Strategies in a Microservice Architecture. Disponível em: <http://martinfowler.com/articles/microservice-testing>. Acesso em: 17 fev. 2016.

The Twelve-Factor App. Disponível em: <http://12factor.net>. Acesso em: 13 mai. 2015.

THONES, Johannes. Microservices. **Software, IEEE**, v. 32, n. 1, p. 116-116, 2015.

VERNON, Vaughn. **Implementing domain-driven design**. 1 ed. Westford: Addison-Wesley, 2013.

VIENNOT, Nicolas et al. Synapse: a microservices architecture for heterogeneous-database web applications. In: **Proceedings of the Tenth European Conference on Computer Systems**. ACM, 2015. p. 21.

VMware buys into Docker containers. Disponível em: <http://www.zdnet.com/article/vmware-buys-into-docker-containers/>. Acesso em: 16 fev. 2016.