
**UNIVERSIDADE FEDERAL DO CEARÁ**
**CAMPUS DE QUIXADÁ**
**ENGENHARIA DE SOFTWARE**

# UM MÉTODO PARA O DESENVOLVIMENTO DE SOFTWARE BASEADO EM MICROSERVIÇOS

**Projeto de Pesquisa**

**Thiago Pereira Rosa**

**Orientadora:**
**Msc. Ticiana Linhares Coelho da Silva**
**Coorientador:**
**Dr. Flávio Rubens de Carvalho Souza**

QUIXADÁ - CE
Julho, 2015

**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, 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 e complexas, a arquitetura monolítica se torna um obstáculo ao desenvolvimento e implantação, além de dificultar a utilização de uma entrega contínua e limitar a adoção de novas tecnologias. Para aplicações complexas é mais interessante utilizar uma arquitetura de microserviços, que divide a aplicação em um conjunto de serviços.

O termo *Microservices* ou Microserviços surgiu nos últimos anos para descrever uma maneira particular de conceber aplicações de *software* como uma suite de serviços independentemente implementáveis (FOWLER; LEWIS, 2014). Embora 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.

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

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

## 2. TRABALHOS RELACIONADOS

Esta seção descreve os trabalhos relacionados que se instituem como estimulo para realização desta proposta.

### *2.1. On Micro-services Architecture*

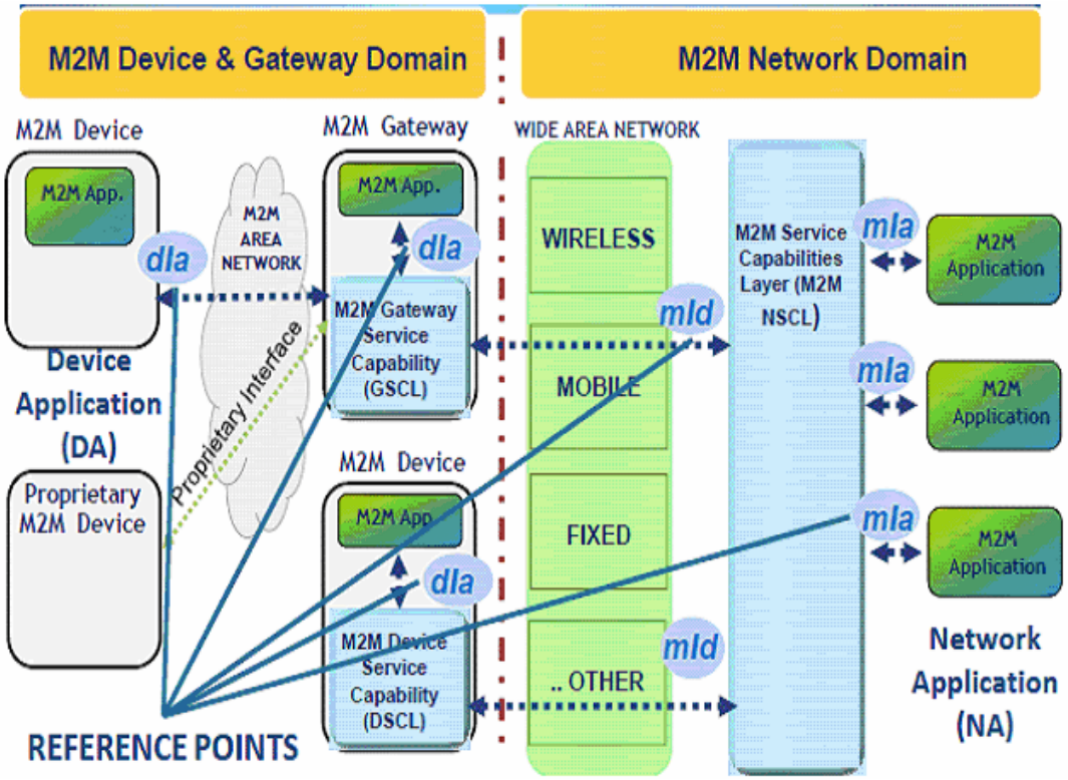
Namiot e Sneps-Sneppe (2014) descrevem a implementação e visão geral de uma arquitetura baseada em microserviços, abordando o fato de que o desenvolvimento deve tornar os serviços leves, independentes e executores de funções simples colaborando entre si por meio de interfaces bem definidas. Foram identificados os princípios comuns utilizando-se esta abordagem, sendo possível avaliar as vantagens e desvantagens do uso da arquitetura de microserviços 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).

Este trabalho também apresenta algumas limitações do uso dos microserviços. 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 inter-serviços e de transações distribuídas.

Além disso, os microserviços contribuem para o aumento de 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 microserviços. Uma abordagem planejada pelos autores foi particionar os serviços em casos de uso, que será exemplificado posteriormente na Figura 2, seguindo os princípios definidos pelo M2M ETSI, mostrado abaixo na Figura 1.

Figura 1 - Modelo *European Telecommunications Standards Institute* M2M



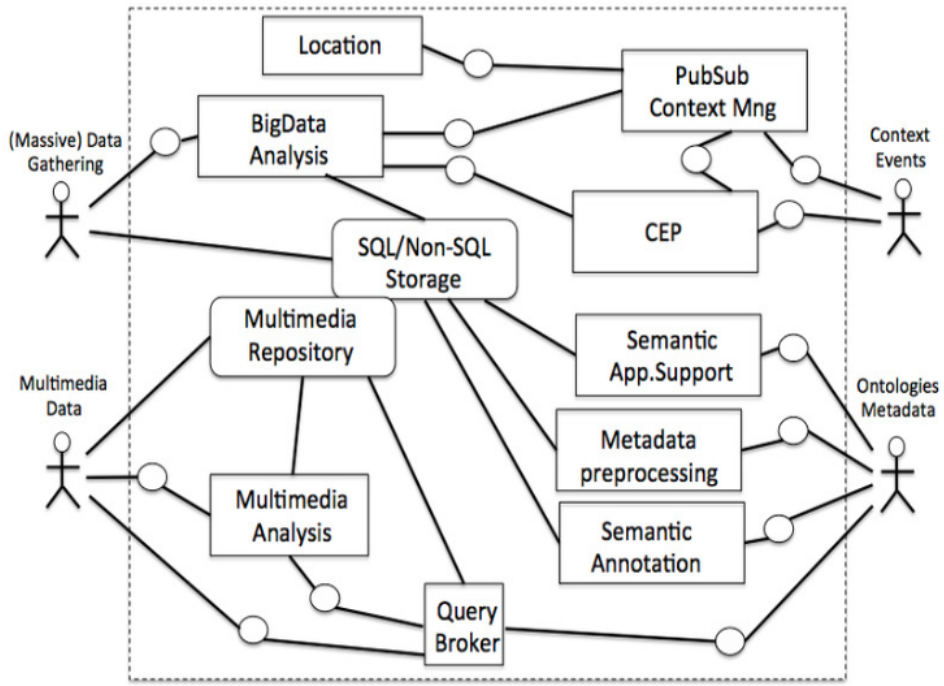

Fonte: ETSI M2M (2014).

Na Figura 1, 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 um dispositivo centralizador e, posteriormente, podem ser roteadas para um sistema computacional.

Outra estratégia é o particionamento por verbos e substantivos, exemplificado abaixo na Figura 2, 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 Responsability Principle* ou Princípio da Responsabilidade Única.

Figura 2 - *FI-WARE data model*, particionado em verbos e substantivos.




Fonte: ELMANGOUSH, AL-HEZMI, MAGEDANZ (2013).

De forma semelhante, a abordagem proposta neste projeto de pesquisa também utilizará uma prática de desenvolvimento focada em sistemas distribuídos com o emprego de testes unitários e um conceito de Contêiner, posteriormente explanado no tópico específico 5.6, para escalar os microserviços da aplicação, além de também implementar um mecanismo de comunicação inter-serviços.

## *2.2. 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 microserviços 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 *layouts* e motores podem ser distintos, porém, devem incorporar as visões compartilhadas entre serviços. Isso permite que os bancos de dados operem em diferentes esquemas, com índices, *layouts* e motores.

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 extendendo 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.

*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, incluíndo MySQL, PostgreSQL, Oracle, MongoDB, Cassandra, entre outros, como pode ser visto na Figura 3.

Figura 3 - 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 modeling |

Fonte: VIENNOT et al. (2015).

Nele foi implementado um conjunto de microserviços que servirão como referência em tecnologia de desenvolvimento e linguagem de programação para a base do método

proposto nesta pesquisa 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.

## 3. FUNDAMENTAÇÃO TEÓRICA

A seguir são apresentados os conceitos-chave que adotados neste trabalho. Nas três primeiras sub-seções são explanados os conceitos de microserviços e, posteriormente, seus princípios, os modelos arquiteturais tradicionais, os modelos arquiteturais baseados em microserviços, *design* orientado a domínio e computação em nuvem, respectivamente. Na sub-seção seguinte é introduzido o conceito de arquitetura de *software*, bem como são tecidas algumas considerações sobre o processo de desenvolvimento de *software* baseado em microserviços.

### 3.1. Microserviços

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

Microserviços, 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, devendo ser facilmente compreendido, modificável e substituível.

O princípio da responsabilidade única, ou SOLID, deve se basear em requisitos funcionais, não funcionais, ou, como o autor aborda, em requisitos multifuncionais. Um exemplo de funcionamento de um microserviço pode ser um processador estar lendo uma mensagens de uma fila na memória, realizando 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 microserviços 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 microserviços é uma abordagem para o desenvolvimento de uma aplicação única, baseada em um conjunto de microserviços, 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.

Figura 4 - Estilo arquitetônico básico baseado em microserviços


Fonte: Elaborado pelo autor.

Observando-se a Figura 4, a Interface do lado do cliente consiste em páginas HTML e Javascript em execução através de um navegador na máquina do usuário, geralmente, um sistema gerenciador de banco de dados (SGBD) relacional. A aplicação do lado servidor lida com as solicitações HTTP, executa a lógica de domínio, recupera e atualiza os dados na base de dados e seleciona e disponibiliza as visões HTML enviadas ao *browser*.

Para Fowler e Lewis (2014), o termo microserviços surgiu nos últimos anos para descrever uma maneira peculiar de desenvolver aplicações de *software* independentemente implementáveis, como uma suite 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 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* independente, substituível e atualizável. Uma aplicação baseada em microserviços pode utilizar bibliotecas, mas o intuito principal é modularizar o *software* em serviços. Para os autores, bibliotecas são componentes que estão ligados a um determinado programa que são chamados utilizando 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 microserviços é minimizar a necessidade de re-implementação, caso ocorram mudanças, por meio da limitação de serviços devidamente coesos com os mecanismos projectados de evolução das interfaces do serviço.

### 3.1.1. Princípios dos Microserviços

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

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

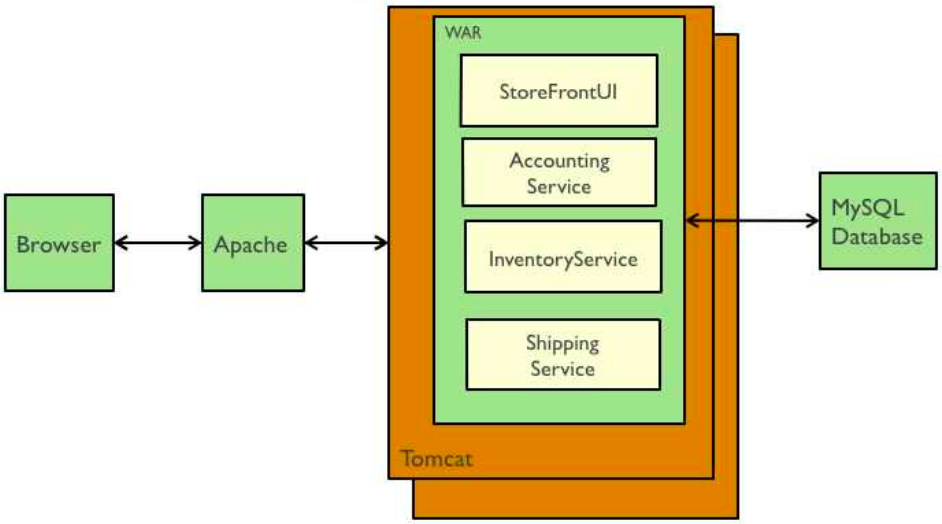
Os princípios abordados acima visam, segundo Newman (2015), reduzir a inércia, fazendo escolhas que favoreçam o *feedback* e mudanças rápidas, reduzindo as dependências entre as equipes. Para eliminar a complexidade acidental, deve-se substituir os processos complexos, desnecessários e principalmente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

### 3.1.2. Modelo de arquitetura *web* tradicional

À primeira vista, o contexto de uma aplicação *web* tradicional possui algumas vantagens em relação ao contexto baseado em microserviços, 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, obtem-se a 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.

Na Figura 5, 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 executado em um contêiner *web*, como neste exemplo do Apache Tomcat[1], um *software* 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.

Figura 5 - Arquitetura tradicional de uma aplicação *web*




Fonte: Microservices architecture (2015).

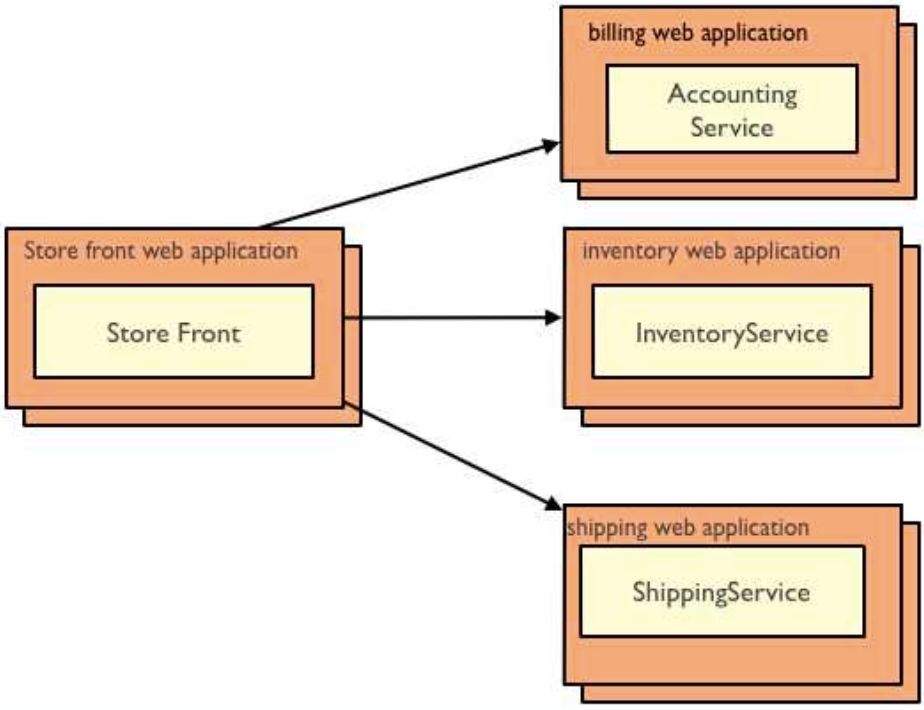
[1] http://tomcat.apache.org

### 3.1.3. Modelo de arquitetura baseada em microserviços

No contexto de microserviços, as vantagens apresentadas no tópico 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 dimensionada com o volume crescente de dados e principalmente não é possível dimensionar cada componente de maneira independente.

A Figura 6 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.

Figura 6 - Aplicação utilizando arquitetura baseada em microserviços




Fonte: Microservices architecture (2015).

## 3.2. 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 no tópico 5.4 sobre a comunicação dos microserviços.

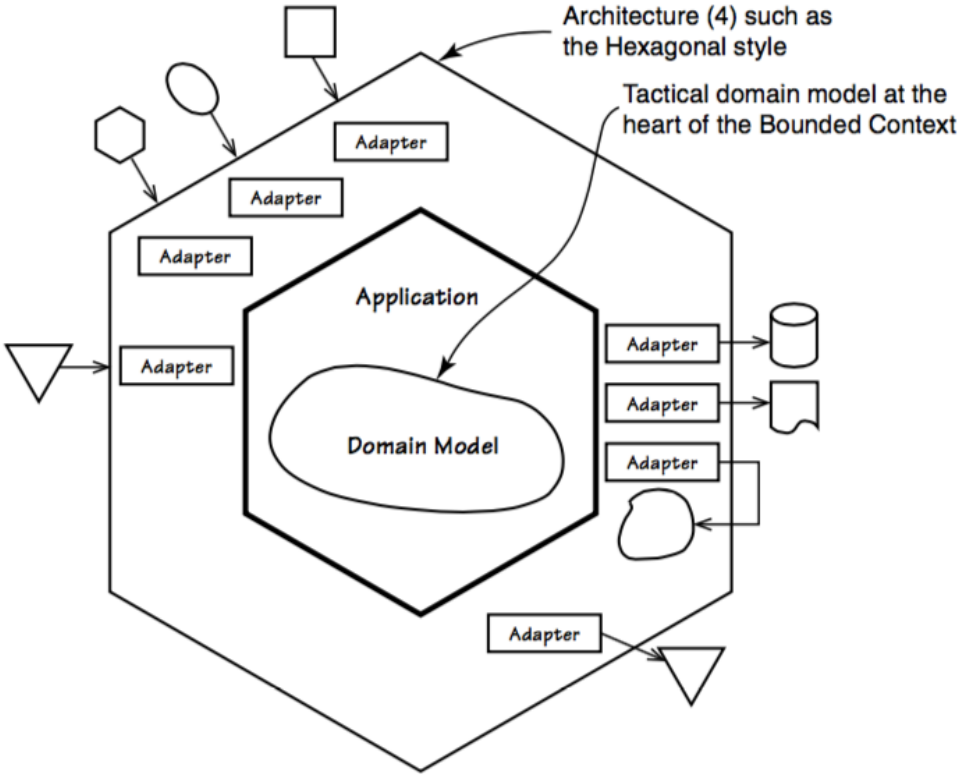
A utilização de boas práticas de desenvolvimento tende a reduzir o tempo necessário para se construir um sistema como um todo; no sentido de desenvolver uma aplicação com processamento distribuído e interconectado, o REST (*Representational State Trasfer*) pode ser utilizado como um modelo arquitetural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

A Figura 7 descreve uma arquitetura hexagonal, definindo o modelo de domínio como o centro do *software*, com sua interface bem definida para o consumo dos serviços por clientes distintos, simples de se empregar e essencialmente focada no modelo de domínio.

Figura 7 - Abstração da arquitetura hexagonal




Fonte: VERNON (2013).

### 3.3. Componentização dos microserviços

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

- 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 serelização de mensagem deve ser conectável (JSON, XML).

### 3.3.1. *Design* orientado a domínio

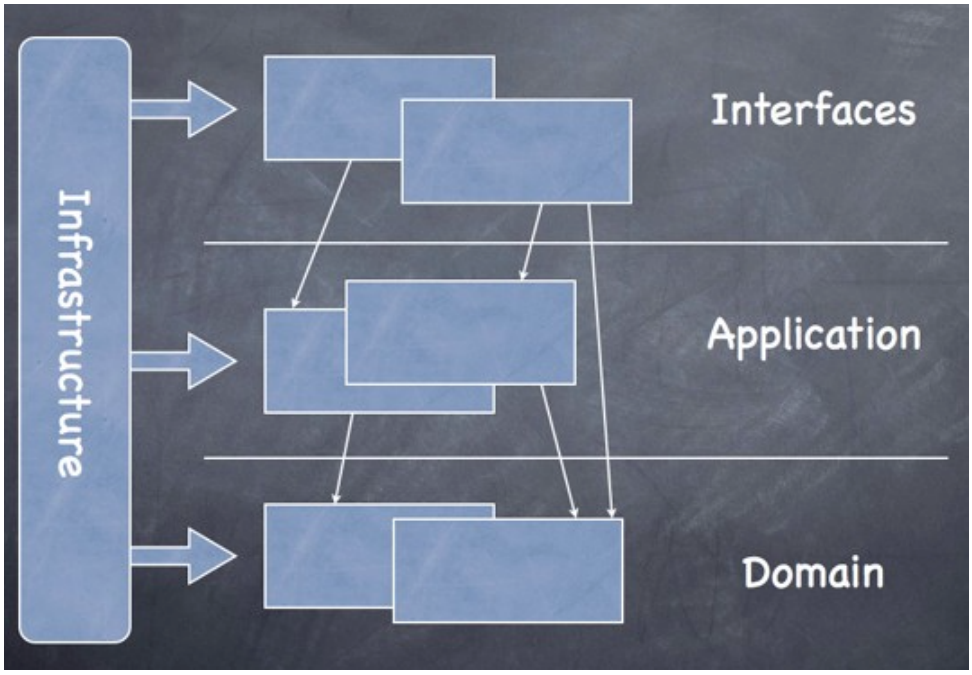
Evans (2015) ajuda a entender a importância de representar as características do mundo real em código, visto que o *Domain Design Driven* ou DDD é um tema amplo e que contém detalhes de difícil incorporação em uma base de código. Sendo assim, deve-se considerar os conceitos em que a lógica de negócio 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 nos 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 nos permite dispor e redimensionar máquinas segundo nossa necessidade, 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), estamos acostumados a pensar em responsabilidades em grupos.

Figura 8 - Ilustração do exemplo do modelo de camadas




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

Na Figura 8, 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.

## 3.4. 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.

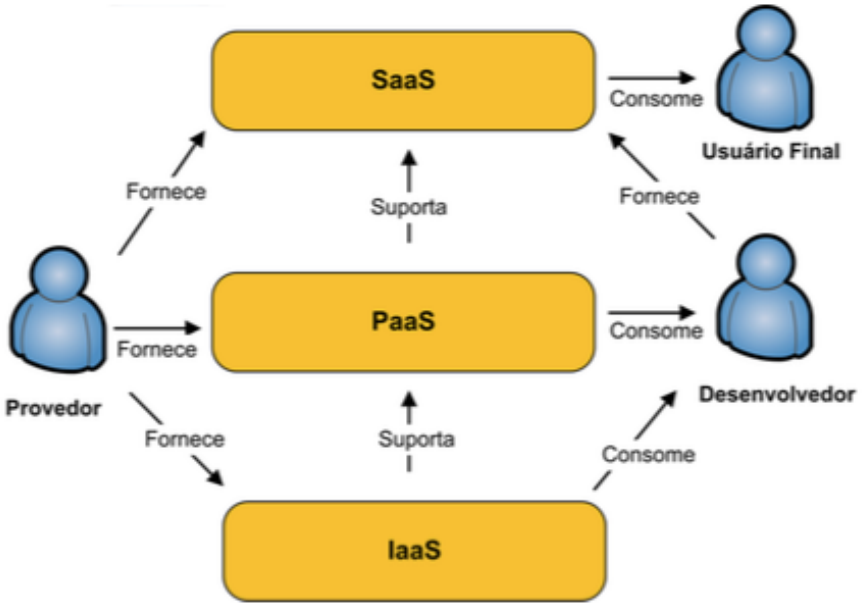
[2] http://dddsample.sourceforge.net/architecture.html

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 pode ser facilmente medida. Os recursos de TI são fornecidos como um serviço, permitindo ao usuário final consumir o serviço 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 9, ou seja, papeis na computação em nuvem para facilitar o entendimento da tecnologia:

Figura 9 - Papéis na computação em nuvem




Fonte: SOUSA, MOREIRA, MACHADO (2009).

- **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.

A computação em nuvem reforça a ideia de que a computação e a comunicação estão fortemente entrelaçadas. Uma das desvantagens da mesma é que ela não pode 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.

## 3.5. 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 aprimoração.

## 4. PROPOSTA

Este trabalho propõe a definição de um método para o desenvolvimento de *software* baseado em microserviços. Este método facilita a construção destes serviços, pois fornece um conjunto de passos bem definidos e utiliza os principais conceitos para desenvolvimento de aplicações distribuídas, utilizando alta tecnologia, computação em nuvem e a habilidade de escala vertical, tendo seus recursos computacionais garantidos, isolados e seguros a fim de evitar a sobrecarga das máquinas físicas devido ao alto número de acessos simultâneos dos multi-usuários. Com a utilização do método de desenvolvimento baseado em microserviços proposto nesta pesquisa se pretende 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 microserviços, evitando custos não planejados com licenciamento ou propriedade intelectual de *software*.

## 5. PROCEDIMENTOS METODOLÓGICOS

Primeiramente, será realizada uma revisão bibliográfica para identificação dos modelos de desenvolvimento baseados em microserviços existentes. Serão identificadas as características e os princípios utilizados para o desenvolvimento de *software* baseado em microserviços. Serão estudas as ferramentas e tecnologias com suporte ao desenvolvimento baseado em microserviços. Com esses conhecimentos, será proposto um método de desenvolvimento no contexto de microserviços. E, por fim, realizaremos um estudo de caso utilizando o método proposto.

### 5.1. Revisão bibliográfica

Este projeto de pesquisa científica se iniciará com a fase de revisão bibliográfica, cujo foco é o desenvolvimento de competências sobre os seguintes temas:

- Microserviços
- Computação em Nuvem
- Virtualização.

### 5.2. Estruturar logicamente os artefatos da solução

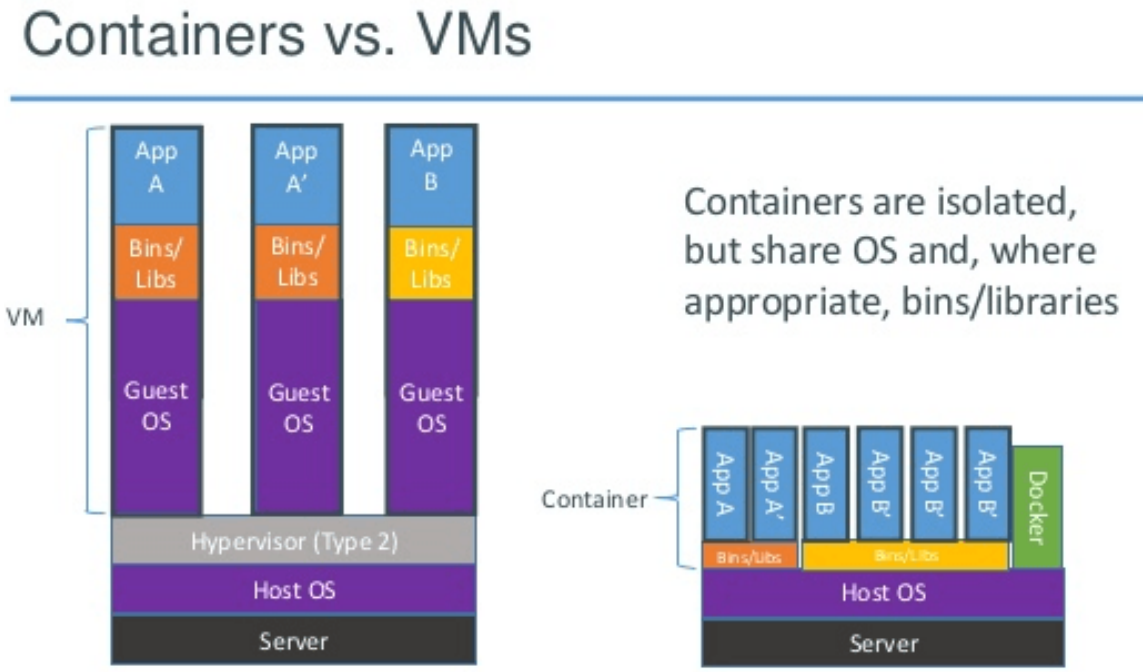
O planejamento e a definição hierárquica da árvore de diretórios da solução têm como intuito facilitar a organização lógica dos artefatos gerados antes, durante e após os processos que permeiam o desenvolvimento e entrega do *software*, enfocando, como objetivo principal, a separação de interesses e 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 microserviços desenvolvidos.

### 5.3. Definir tecnologias para auxiliar na construção de microserviços

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



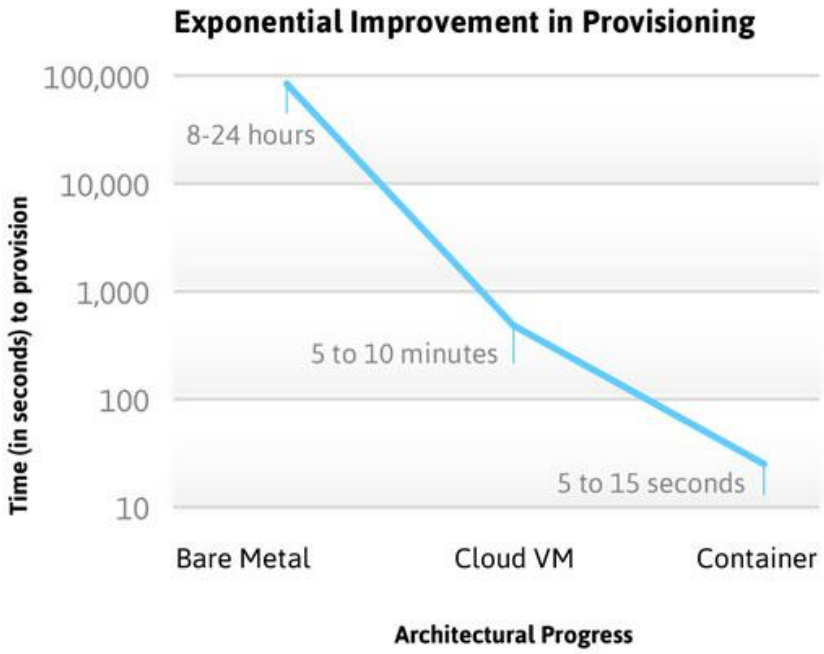

Fonte: Diferença entre tecnologias de virtualização e contêiner[3]

Como pode 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* como um todo.

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, não levando em consideração 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.

[3] http://www.zdnet.com/article/vmware-buys-into-docker-containers/

Figura 11 - Provisionamento das tecnologias de VM tradicionais e contêineres




Fonte: Melhoria no provisionamento de recursos[4]

A Figura 11 mostra a comparação em tempo na escala de milissegundos, com base no progresso arquitetural, do 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, é conhecida como Tipo 1, em que o sistema operacional se comunica diretamente com o *hardware*.

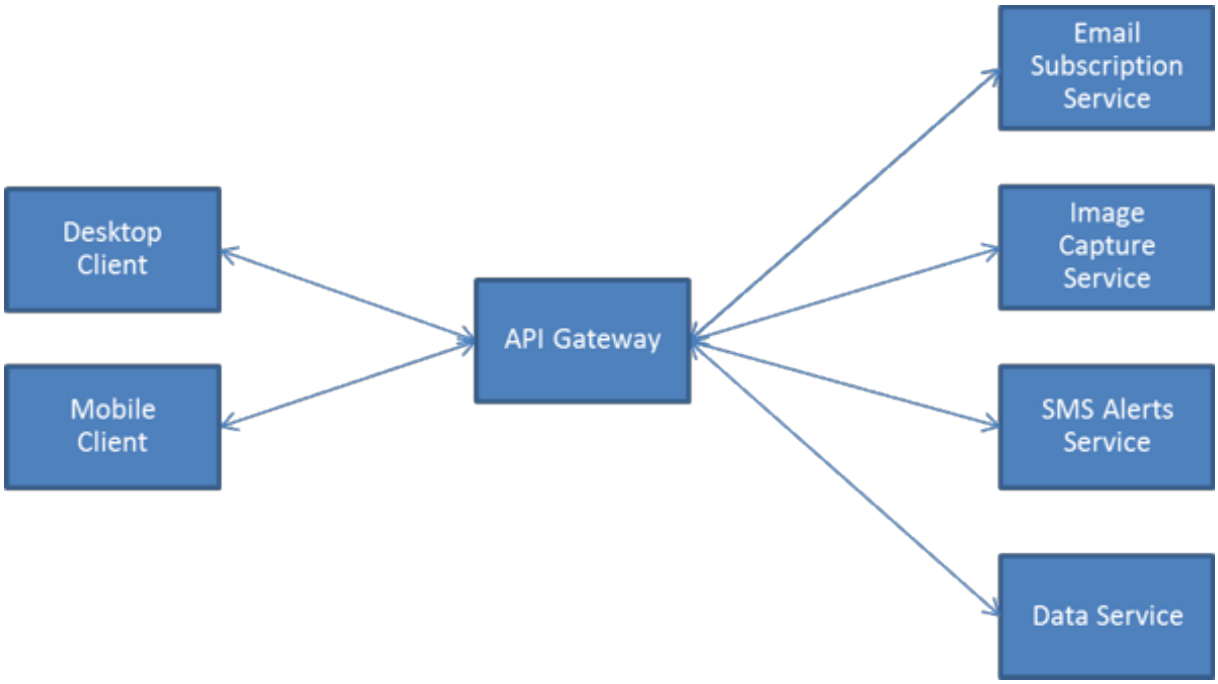
## 5.4. Planejar a comunicação dos microserviços

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

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

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

Figura 12 - Uma abordagem para comunicação dos microserviços




Fonte: Abordagem para comunicação centralizada dos microserviços[5]

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

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

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

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

único processo para servir a um grande número de pedidos, não bloqueando as operações de entrada.

Abaixo são elencadas as características básicas principais, citadas pelo autor, de sistemas baseados em processos e em eventos.

Sistemas baseados em Processos:

- Processo simples / *Threads* por pedido.
- I/O Bloqueante.
- Conjunto encadeado de processos.

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 nos permitirá adotar o estilo correto para planejar a comunicação dos microserviços. Balla auxilia nossa 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.

## 5.5. Definir método para construir microserviços

Esta etapa consiste em definir o método para a construção dos microserviços. Para tanto, este trabalho irá se basear no manifesto denominado *The Twelve-Factor* que disponibiliza uma metodologia semi formal para construção de produtos de *software* como um serviço, defendendo as seguintes premissas:

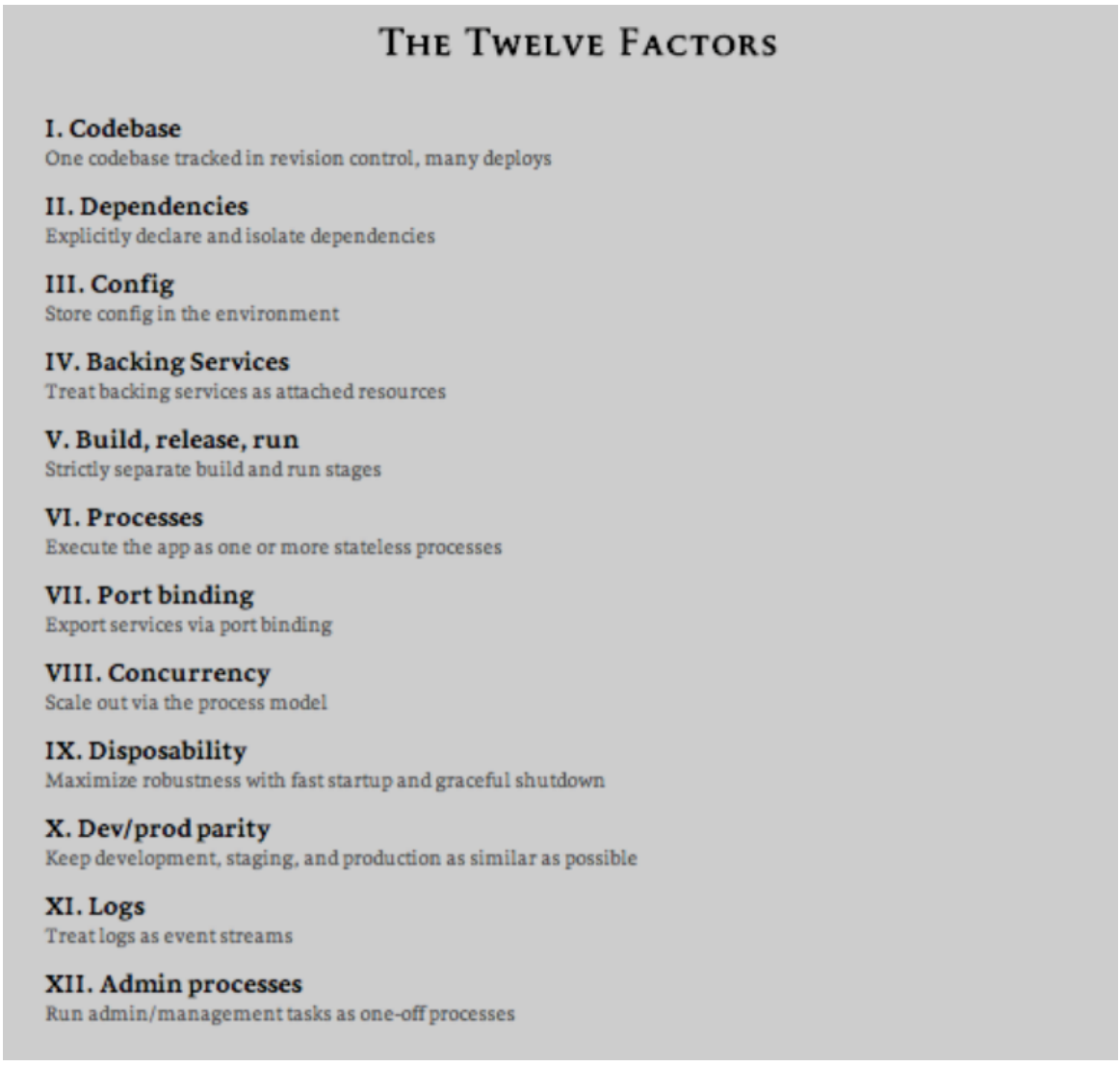
- Utilizar formatos declarativos para automação de configuração.
- Possuir contrato (*interface*) bem definido com o sistema operacional subjacente.
- Ser adequado para a implantação em plataformas de nuvem modernas.
- Minimizar a divergência entre desenvolvimento e produção.
- Possibilitar a escalabilidade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

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. O foco do manifesto está nos desenvolvedores e engenheiros que implantam ou administram *software* como um serviço, abordando assuntos como controle de versão, ambiente, construção, comunicação e a execução de tarefas de gestão e administração.

Figura 13 - Princípios apoiados pelo The Twelve-Factors




Fonte: The Twelve-Factors[6]

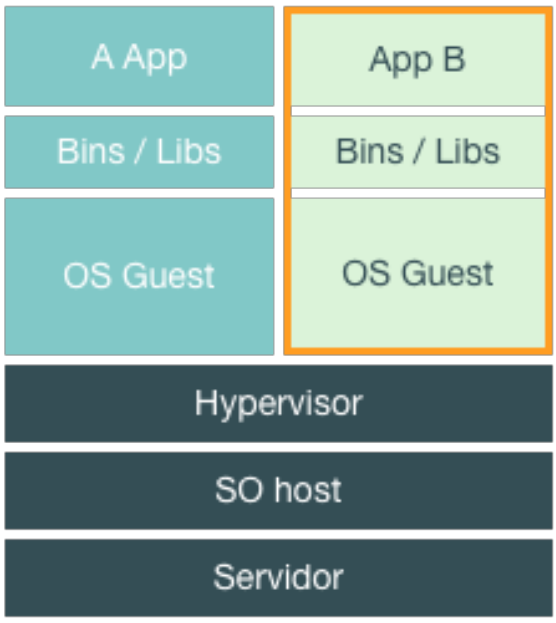
[6] http://12factor.net

## 5.6. Implantar os microserviços construídos

Uma das vantagens em se utilizar os microserviços 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 - Máquina Virtual



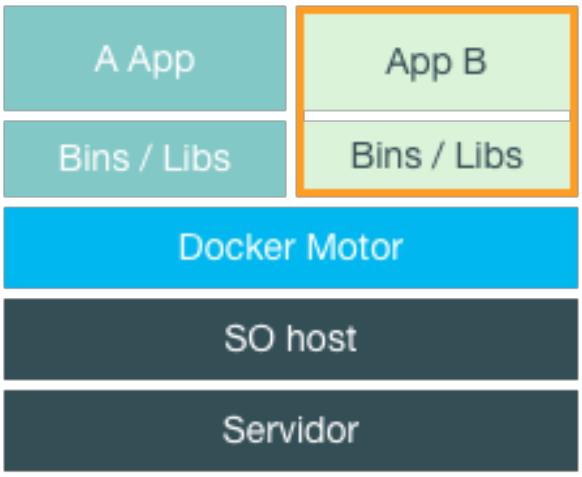

Fonte: Desmembramento dos recursos de uma máquina virtual tradicional[7]

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

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

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 *Gigabyte* (GB) de armazenamento em disco.

Figura 15 - *Docker*




Fonte: Divisão em camadas de uma tecnologia de contêiner[8]

Na Figura 15, 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.

## 5.7. Realizar um estudo de caso

O estudo de caso consistirá em utilizar o método proposto nesta pesquisa para desenvolver um sistema para auxiliar o tratamento médico com o uso de fototerapia. Este sistema será composto por um conjunto de serviços independentes, além de acesso via dispositivos móveis.

A partir da conclusão desse estudo de caso será 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, destacar as forças e fraquezas do método no cenário e contexto em que o mesmo foi aplicado.

---

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

## 5.8. Cronograma de execução

| ATIVIDADES | 2015 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Mai | Jun | Jul | Ago | Set | Out | Nov | Dez |
| Aprofundamento da revisão bibliográfica | X | X | | | | | | |
| Definir o método proposto | | X | X | X | X | | | |
| Definir tecnologias de desenvolvimento | | | X | X | | | | |
| Desenvolvimento do estudo de caso | | | | X | X | X | | |
| Testes | | X | X | X | X | X | | |
| Análise dos resultados | | | | | | X | | |
| Criação e revisão da monografia | | | | X | X | X | | |
| Defesa | | | | | | | X | |

## 6. REFERÊNCIAS


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.

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

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.

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.

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

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.

**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.