
**MESTRADO EM SISTEMAS E COMPUTAÇÃO**

**HUGO HENRIQUE OLIVEIRA SAMPAIO DA SILVA**

# UM GUIA PARA APOIAR A MIGRAÇÃO DE SISTEMAS DE SOFTWARE LEGADOS PARA ARQUITETURA BASEADA EM MICROSERVIÇOS

Salvador
2018

**HUGO HENRIQUE OLIVEIRA SAMPAIO DA SILVA**

# UM GUIA PARA APOIAR A MIGRAÇÃO DE SISTEMAS DE SOFTWARE LEGADOS PARA ARQUITETURA BASEADA EM MICROSERVIÇOS

Dissertação apresentada ao Programa de Pós-Graduação da UNIFACS Universidade Salvador, Laureate International Universities como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Glauco de Figueiredo Carneiro.

Salvador
2018

FICHA CATALOGRÁFICA
(Elaborada pelo Sistema de Bibliotecas da UNIFACS Universidade Salvador, Laureate International Universities)

Silva, Henrique Oliveira Sampaio da

Um guia para apoiar a migração de sistemas de software legados para arquitetura baseada em microserviços./ Hugo Henrique Oliveira Sampaio da Silva. – Salvador: UNIFACS, 2018.

71 f. : il.

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da UNIFACS Universidade Salvador, Laureate International Universities como parte dos requisitos para a obtenção do título de Mestre.
Orientador: Prof. Dr. Glauco de Figueiredo Carneiro.

1. Migração de software. I. Carneiro, Glauco de Figueiredo, orient. II. Título.

CDD: 004.6

**TERMO DE APROVAÇÃO**

**HUGO HENRIQUE OLIVEIRA SAMPAIO DA SILVA**

**UM GUIA PARA APOIAR A MIGRAÇÃO DE SISTEMAS DE SOFTWARE LEGADOS PARA ARQUITETURA BASEADA EM MICROSERVIÇOS**

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Ciência da Computação, UNIFACS Universidade Salvador, Laureate International Universities, apresentada à seguinte banca examinadora:

Glauco de Figueiredo Carneiro – Orientador _____
Pós DFoutor pela University of Wisconsin-Milwaukee (EUA)
UNIFACS Universidade Salvador, Laureate International Universities

Eduardo Magno Lages Figueiredo _____
Doutor em Engenharia de Software pela Lancaster University
Universidade Federal de Minas Gerais – UFMG

Ana Patrícia Fontes Magalhães Mascarenhas _____
Doutora em Ciências da Computação pela Universidade Federal da Bahia - UFBA
UNIFACS Universidade Salvador, Laureate International Universities

Salvador, 31 de Agosto de 2018.

## AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus por me proporcionar mais esta realização. Aos meus pais, por serem fonte de inspiração e meus maiores tesouros. Ao meus irmãos e minha avó por todo amor e carinho. À toda minha família por terem contribuído de maneira significativa no meu caráter. Sem vocês, não teria chegado até aqui.

*“A melhor maneira de prever o futuro é criá-lo.”*

*(Peter Druker)*

## RESUMO

*Contexto:* A literatura fornece evidências sobre os desafios e dificuldades relacionadas à migração de sistemas de software monolíticos legados para arquiteturas baseadas em microserviços. O principal conceito por trás dessas arquiteturas é a transformação do software em um conjunto de pequenos serviços, de implementação independente nos quais cada serviço é executado em seu próprio processo e a comunicação é realizada através de mecanismos leves e bem definidos para atender ao objetivo de negócios. No entanto, a literatura carece de um guia que auxilie a realização de uma migração do ponto de vista do praticante. *Objetivos:* Propor um roteiro para auxiliar profissionais da indústria e da academia a migrar sistemas de software legados para a arquitetura baseada em microserviços. Além disso, analisar até que ponto o roteiro proposto no formato de um guia viabiliza a identificação de funcionalidades candidatas que podem ser moduladas no contexto de aplicações legados para posteriormente serem convertidos em microserviços. *Metódos:* Foram realizados dois estudos exploratórios. O primeiro um estudo piloto, com o objetivo identificar etapas relevantes e eficazes de um roteiro preliminar, para apoiar a migração de sistemas de software monolítico legado para uma arquitetura baseada em microserviços. E o segundo um estudo de caso, que aprimorou o roteiro preliminar proposto no estudo piloto. *Resultados:* A elaboração de um guia que propoem auxiliar profissionais durante a etapa de migração de sistema de software monolítico legado para arquitetura baseada em microserviços. E também a divulgação das lições aprendidas com a execução dos estudos exploratórios. *Conclusão:* Este trabalho fornece um conjunto de etapas organizadas como um guia central com base na experiência adquirida durante a execução da migração de dois sistemas legados.



**Palavras-chaves**: Sistemas Legados Monolíticos. Estudo Exploratório. Microserviços.

## ABSTRACT

*Context:* The literature provides evidence on the challenges and difficulties related to the migration of legacy monolithic software systems to microservice-based architectures. The main concept behind these architectures is the transformation of software into a set of small services, independent implementation in which each service is executed in its own process and communicates through well-defined, lightweight mechanisms to serve a business goal . However, the literature lacks a step by step guideline that helps to carry out a migration from the practitioner's point of view.*Objectives:* Propose a roadmap to support practitioners to migrate legacy software systems to microservice-based architecture. In addition, we analyze the effectiveness of a guide in the search for candidate functionalities that can be modulated in the context of legacy applications and later be converted into microservices. *Methods:* Two exploratory studies were carried out. The first was a pilot study, aiming to identify relevant and effective steps of a preliminary roadmap, to support the migration of legacy monolithic software systems to a microservice-based architecture. And second a case study, which improved the preliminary roadmap, proposed in the pilot study chapter. *Results:* The elaboration of a guide that proposes to support practitioners during the migration step from monolithic legacy software to architecture based on microservices. And also the dissemination of lessons learned from exploratory studies. *Conclusion:* This work provides a set of steps organized as a central guide based on the experience obtained during the execution of migration of two legacy software systems.



**Keywords**: Monolithic Legacy Systems. Exploratory Study. Microservices.

## LISTA DE FIGURAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| BPM | Business Process Management |
| COS | Computação Orientada a Serviços |
| DDD | Domain-Driven Design |
| ESB | Enterprise Service Bus |
| TI | Tecnologia da Informação |
| SOC | Service-Oriented Computing |
| SOA | Service-Oriented Architecture |
| SOLID | Single Responsibility, Open-closed, Liskov Substitution, Interface Segregation and Dependency Inversion |
| SOAP | Simple Object Access Protocol |
| SPA | Single-Page Applications |
| URL | Uniform Resource Locator |
| UDDI | Universal Description Discovery and Integration |
| XML | Extensible Markup Language |
| WSDL | Web Services Description Language |

**SUMÁRIO**

# 1 INTRODUÇÃO

Este capítulo apresenta a contextualização, motivação problemas abordados, objetivos, etapas adotadas nesta dissertação, contribuições e a estrutura dos próximos capítulos.

## 1.1 CONTEXTO

Microserviços representam um arcabouço arquitetural inspirado em uma abordagem orientada a serviços (DRAGONI et al., 2017). Este arcabouço representa uma solução promissora para a construção eficiente e gerenciamento de sistemas de software complexos (SINGLETON, 2016). A adoção de uma arquitetura baseada em microserviços tem o objetivo de reduzir custos, melhorar a qualidade, agilizar e diminuir o tempo de desenvolvimento de um sistema para a disponibilização para o mercado.

Os microserviços podem ser considerados como o equivalente de software dos brinquedos Lego: eles funcionam quando integrados e encaixados adequadamente e podem ser uma opção para construir soluções complexas em menos tempo do que com soluções arquitetônicas tradicionais (SINGLETON, 2016).

Muitos sistemas de software legados foram movidos para a nuvem sem ajustes prévios em sua arquitetura para a nova infraestrutura. Muitos deles foram originalmente colocados em máquinas virtuais e implantados na nuvem, assumindo as características de recursos e serviços de um *data center* tradicional. Essa abordagem não reduz custos, nem melhora o desempenho e a manutenção (TOFFETTI et al., 2017).

Uma questão ainda em aberto está relacionada às etapas que devem ser seguidas para migrar um sistema legado monolítico para uma arquitetura baseada em microser- viços. Existe esta discussão em alguns trabalhos (KALSKE; MäKITALO; MIKKONEN, 2017) (LEYMANN et al.,2016) (TAIBI; LENARDUZZI; PAHL,2017). Para apresentar uma proposta para esta questão, esta dissertação apresenta um conjunto de etapas organi- zadas como um guia central com base na experiência adquirida

durante a migração de dois sistemas legados.

O guia é o resultado de um estudo de duas fases para abordar a seguinte: **Questão de Pesquisa (QP)**: *Quais etapas podem ser adotadas para migrar um sistema de software legado para uma arquitetura baseada em microserviços?* A disponibilidade de um conjunto de etapas organizadas em um guia de migração de sistemas legados para microserviços para apoiar profissionais da indústria e da academia pode ser uma oportunidade para incentivá-los a aceitar esse desafio.

## 1.2 ABORDAGEM DE PESQUISA

Nesta seção são elicitados os problemas, objetivos, metodologia e contribuições.

### 1.2.1 Problemas da Pesquisa

O problema de pesquisa que motiva este trabalho é a dificuldade de encontrar na literatura um guia para orientar como identificar de maneira prática os limites de serviços e candidatos a serviços de forma coesa, bem como a migração de um sistema monolítico legado para microserviços.

### 1.2.2 Objetivos da Pesquisa

**Objetivo Geral (OG)**: Propor um roteiro para auxiliar profissionais da indústria e da academia a migrar sistemas de software legados para a arquitetura baseada em microserviços.

**Objetivos Específtcos (OE)**

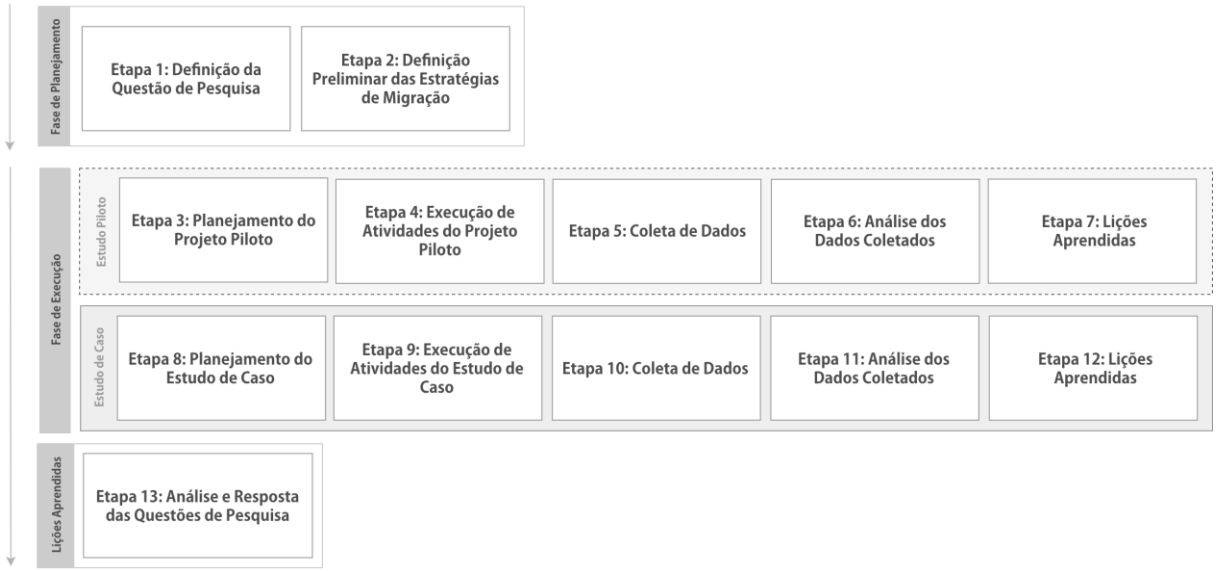
*Objetivo Específico (OE1)*: Identificar os principais desafios enfrentados durante a migração de um sistema de software legado.

*Objetivo Específico (OE2)*: Analisar até que ponto um conjunto de etapas apoiaria na busca de funcionalidades candidatas que poderiam ser moduladas no contexto de aplicações legadas para posteriormente serem convertidas em microserviços.

*Objetivo Específico (OE3)*: Buscar evidências que apresentem vantagens na

utilização de conceitos e padrões do *Domain-Driven Design* (DDD) na decomposição de um sistema de software legado para microserviços.

Figura 1 – Etapas da Dissertação




## 1.3 ETAPAS DA PESQUISA

A figura 1 ilustra as etapas adotadas para investigar e responder as questões de pequisa estabelecidas. As etapas foram agrupas em três fases: fase de planejamento, fase de execução e as lições apreendidas. Os parágrafos seguintes descrevem cada fase.

### 1.3.1 Fase de Planejamento

A fase de planejamento possibilitou construir um referencial para os caminhos a percorrer, sendo dividida em duas etapas descritas a seguir.

**Definição das Questões de Pesquisa (Etapa 1 da Figura 1)** . As questões de pesquisa foram consolidadas através da revisão da literatura, dos problemas de pesquisa e dos objetivos gerais e específicos identificados anteriormente.

- Questão de Pesquisa 1 (QP1): Como encontrar funcionalidades em um sistema de software legado para posteriormente serem modularizados e adaptadas

à uma arquitetura baseada em microserviços?

- Questão de Pesquisa 2 (QP2): Como migrar funcionalidades candidatas para uma arquitetura baseada em microserviços?

- Questão de Pesquisa 3 (QP3): Quais as etapas mínimas necessárias para compor um guia de migração de sistemas de software legados, para uma arquitetura baseada em microserviços?

**Definição Preliminar das Estratégias de Migração (Etapa 2 da Figura 1)**. Nesta etapa, as estratégias têm o objetivo de orientar e apoiar inicialmente a migração, definindo cada passo a ser executado na tentativa de reconhecer nestes projetos, as principais funcionalidades e suas respectivas responsabilidades. Estas estratégias foram organizadas em três fases. A fase 1 tem o foco na análise e identificação das funcionalidades. A fase 2 detalha as características identificadas no sistema monolítico, usando alguns dos conceitos disponíveis no DDD. Na fase 3 o foco é voltado a decisão e implementação da arquitetura de microserviços.

**Fase de Execução (Etapa 3 da Figura 1)**. A fase de execução é composta das seguintes etapas: Identificar Domínios e Subdomínios das Funcionalidades Identificadas, Identificar os Contextos Delimitados e Construir um Mapa de Contextos.

**Planejamento do Projeto Piloto (Etapa 4 da Figura 1)**. Nesta etapa, foi definido como realizar o reconhecimento dos artefatos do sistema legado para auxiliar na compreensão das funcionalidades oferecidas pelo sistema, e em seguida auxiliar na execução das atividades propostas pelo guia. Foi constatado que, para obter mais conhecimento dos artefatos e compreender o domínio do sistema era necessário extrair os dados a partir do código-fonte. Este foi um processo contínuo, executado de forma manual através da navegação dos arquivos e diretórios do sistema.

Em seguida, após algumas execuções deste processo, optou-se também em analisar e extrair dados a partir da base de dados do sistema.

**Execução de Atividades do Projeto Piloto (Etapa 4 da Figura1)**. Nesta etapa, foi executada as atividades de identificação e reconhecimento dos relacionamentos entre os artefatos, para em seguida identificar as funcionalidades do sistema. Com

base nos dados obtidos após a identificação das funcionalidades foi construído um mapa de contexto com o objetivo de identificar e validar os limites entre os recursos, e também entender o nível de acoplamento do código-fonte com a estrutura da aplicação.

A análise dos artefatos foi realizada de forma manual em cada um dos artefatos reconhecidos. Após à análise foi possível identificar e reconhecer a necessidade de refatoração, e portanto, foi executada uma mudança estrutural da aplicação, aplicando conceitos e padrões de projetos que apoie minimizar o acoplamento do código emaranhado.

**Coleta de Dados (Etapa 5 da Figura 1)**. A coleta de dados consistiu na obtenção de dados do código-fonte e da estrutura de arquivos e diretórios do projeto para viabilizar a elaboração dos seguintes artefatos (*Entities*, *Value Object's*, *Repositories*, *Aggregate Root's*, *Domain Services* e *Factories*).

**Análise dos Dados Coletados (Etapa 6 da Figura1)** . A análise dos dados teve como fonte, o código-fonte dos projetos e os artefatos construídos indicados na etapa de coleta de dados.

**Lições Aprendidas (Etapa 7 da Figura 1)**. A partir da análise de dados, foi identificado através da experiência da realização do estudo, o que pode ser melhorado e qual a relevância das etapas realizadas para responder as questões de pesquisa do estudo. As lições aprendidas também servem para identificar até que ponto as respostas de questões de pesquisas do estudo podem ser generalizadas e quais são as limitações impostas pelas características do estudo realizado.

As etapas 8, 9, 10, 11 e 12 do estudo de piloto, são as mesmas etapas do estudo de caso.

### 1.3.2 Análise e Respostas das Questões de Pesquisa (Etapa 13 da Figura 1)

A partir das evidências encontradas na literatura tanto através da Revisão Bibliográfica como no resultado do estudo explorátorio, foi possível analisar as questões de pesquisa desta dissertação e consequentemente apresentar respostas para cada uma delas.

### 1.3.3 Resultados Obtidos

Com o estudo piloto foi possível atender apenas a primeira parte do roteiro preliminar proposto. Esta primeira parte do roteiro está relacionada à restruturação de um sistema em sua versão legada monolítica para um versão modularizada. Devido ao cenário simples encontrado no estudo piloto, foi percebido que migrar para uma arquitetura baseada em microserviços iria aumentar a complexidade da migração.

No estudo de caso foi possível responder as três questões de pesquisa desta dissertação, que serão descritas a seguir.

**QPD1**. *Como encontrar funcionalidades em um sistema de software legado para posteriormente serem modularizados e tornar-se adaptados a uma arquitetura baseada em microserviços?* Para encontrar as funcionalidades candidatas do sistema, foi realizado um processo manual iterativo de identificação, navegando entre os arquivos e diretórios do sistema de arquivos do projeto, para compreender melhor o objetivo de cada artefato.

**QPD2**. *Como migrar funcionalidades candidatas para uma arquitetura baseada em microserviços?* Para compreender melhor o domínio do sistema de software, o DDD fornece algumas ferramentas que auxiliam a definição de um serviço. Para reconhecer os limites e as responsabilidades das funcionalidades no sistema, a ferramenta *Bounded Context* (EVANS,2004) foi utilizada, por facilitar a criação de componentes menores e mais coerentes a determinadas áreas de negócio. Após a correta identificação dos contextos do sistema, o *Context Map* é uma técnica de uso geral, que faz parte do conjunto de ferramentas do DDD, auxilia o reconhecimento e a identificação destes contextos em todo o sistema de forma visual. Como apontado em (NEWMAN, 2015), a correta identificação dos contextos limitados e a quebra de um sistema grande entre eles é uma forma eficaz de definir limites de microserviços.

**QPD3**. *Quais as etapas necessárias de um guia para migrar sistemas de software legados para uma arquitetura baseada em microserviços?* Baseado na experiência adquirida durante a execução dos estudos, foi possível sugerir um conjunto de atividades conforme descrito no capítulo 5.

A ordem seguida das atividades: (1) Analisar o Código-Fonte e o Banco de Dados, (2) Identificar as Principais Funcionalidades, (3) Identificar Domínios e Subdomínios das Funcionalidades Identificadas, (4) Identificar os Contextos Delimitados, (5) Construir um Mapa de Contextos, (6) Migrar para a Arquitetura de Microserviços. Após seguir este conjunto de atividades é importante executar testes automatizados, preferencialmente testes unitários para garantir que as unidades modificadas estejam funcionando conforme esperado.

### 1.3.4 Contribuições

A principal contribuição desta dissertação está direcionada na elaboração de uma guia que propoem auxiliar profissionais durante a etapa de migração de sistema de software monolítico legado para arquitetura baseada em microserviços. Outra contribuição é a divulgação das licões aprendidas com a execução dos estudos exploratórios.

## 1.4 ESTRUTURA DESTA DISSERTAÇÃO

**Capítulo 2:** Apresenta a Fundamentação Teórica, assim como conceitos sobre arquitetura monolítica, arquitetura orientada a serviços, de maneira sintetizada descreve as oportunidades proporcionadas pela computação em nuvem, arquitetura baseada em microserviços, containers e devops.

**Capítulo 3:** Descreve um estudo piloto, que é um estudo exploratório composto de duas fases, com o objetivo de identificar etapas relevantes e eficazes de um roteiro preliminar que apoie a migração de sistema de software legado para uma arquitetura baseada em microserviços.

**Capítulo 4:** Apresenta uma versão revisada e aprimorada do roteiro preliminar, proposto no capítulo do estudo piloto, através de um estudo de caso da migração de um sistema de software monolítico e legado de uma loja online para uma arquitetura baseada em microserviços. Ao final, apresentou as lições aprendidas durante a implantação deste estudo de caso relacionando com trabalhos existentes na literatura.

**Capítulo 5:** Apresenta as etapas de um guia proposto para apoiar a migração de sistema de software legado monolítico para arquitetura baseada em microserviços. Este guia está organizado em três fases principais.

**Capítulo 6:** Este capítulo consolida as considerações finais desta dissertação, são destacadas as contribuições e perspectivas para trabalhos futuros.

## 1.5 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou os conceitos desta dissertação, problemas, objetivos, metodologia utilizada, contribuições, além da estrutura que segue a presente dissertação. O capítulo seguinte apresenta o fundamentação teórica, que é essencial para a compreensão desta pesquisa.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos utilizados nesta dissertação. Para esta finalidade, são discutidas as características e limitações das arquiteturas monolítica, orientada a serviços e de microserviços. Em seguida, apresenta as oportunidades ofertadas pela computação em nuvem e de maneira sintetizada aborda o conceito de *container's*. Ao final, apresenta a metodologia DevOps (Desenvolvimento e Operações).

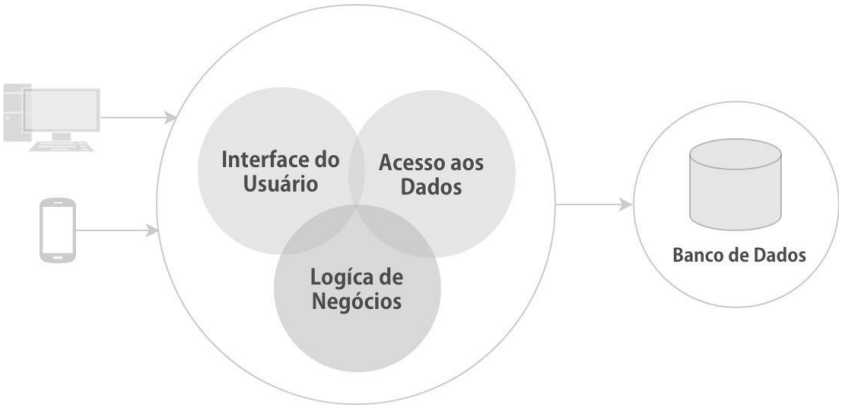
## 2.1 CARACTERÍSTICAS DA ARQUITETURA DE SOFTWARE MONOLÍTICA

Em um sistema com arquitetura monolítica, a maioria dos artefatos são executados em um ambiente centralizado, compartilhando os recursos da mesma máquina (memória, banco de dados e sistema de arquivos) (DRAGONI et al., 2017). O sistema pode interagir com outros serviços ou armazenamentos de dados durante a execução de suas operações, mas o núcleo de seu comportamento é executado em seu próprio processo e o sistema inteiro é normalmente implantado como uma única unidade.

A abordagem monolítica como ilustrado na Figura2, introduz algumas limitações, dentre elas:

*Sistema com Base de Código Grande*. Os sistemas monolíticos com uma base grande de código são difíceis de manter e evoluir devido à sua complexidade (DRAGONI et al., 2017). Um novo desenvolvedor que passe a integrar a equipe, pode ter dificuldades de se familiarizar com o código (SANTIS; ROSA, 2016). O rastreamento de bugs nessas condições exige muito esforço e, portanto, é provável que diminua a produtividade da equipe. Isso também pode fazer com que os desenvolvedores fiquem menos produtivos e desencorajados diante de novas tentativas de alterações (DRAGONI et al., 2017).

Figura 2 – Um Exemplo de Arquitetura Monolítica




*Maior Esforço para a Implantação*. Uma alteração realizada em um sistema monolítico implica em ter que republicar toda aplicação. À medida que o sistema evolui, fica cada vez mais difícil mantê-lo e rastrear sua arquitetura original [1]. Isso pode resultar em interrupções recorrentes, especialmente para projetos de grande porte, dificultando o desenvolvimento, testes e atividades de manutenção (DRAGONI et al.,2017).

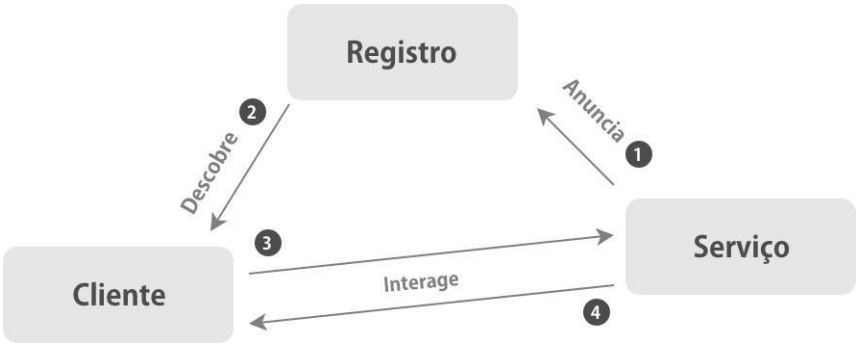
*Dificuldade para Escalabilidade Horizontal*. Se o sistema precisa ser escalado horizontalmente (também chamado de *scale out/in*), que é um processo de incorporar novas máquinas para atender o aumento da demanda pelos usuários finais; geralmente toda a aplicação necessita ser duplicada em vários servidores. Na tentativa de lidar com essa deficiência e também com o crescimento de solicitações de entrada nesses aplicativos, os desenvolvedores criam novas instâncias deles e dividem a carga entre essas instâncias. Infelizmente, esta não é uma solução eficaz, pois o aumento do tráfego é direcionado apenas para um subconjunto dos módulos, causando dificuldades para a alocação de novos recursos em outros componentes (SANTIS; ROSA, 2016) (DRAGONI et al., 2017) (GARCIA et al., 2008).

*Dificuldade para Adotar Novas Tecnologias*. Em uma arquitetura monolítica, a adoção de novas tecnologias é geralmente uma tarefa díficil, devido ao nível de acopla- mento entre as camadas de sistema e a tecnologia adotada. Como resultado, uma aplicação com arquitetura monolítica necessáriamente terá que continuar a usar a tecnologia exis- tente, o que, consequentemente, é um obstáculo para que a aplicação acompanhe novas tendências (SANTIS; ROSA,2016).

## 2.2 ARQUITETURA ORIENTADA A SERVIÇOS

A arquitetura orientada a serviços (SOA - conhecido como Service-Oriented Architecture) é um estilo arquitetônico que promove a criação de serviços orientados a negócios[1]. Cada um dos serviços representa uma funcionalidade específica mapeada explicitamente para uma etapa em um processo de negócio (GROVES, 2005). A abordagem discutida em (ERL, 2008) sugere uma separação técnica e funcional dos serviços.

Figura 3 – Interação de Serviço em um Ambiente Orientado a Serviços




No contexto de SOA, serviço é um componente de software que pode ser reutilizado por outro serviço ou também pode ser acessado atráves de uma interface conectada à rede. A interface de um serviço fornece a identificação, definição dos parâmetros e contrato para transferir os resultados do serviço de volta ao consumidor.

Os serviços podem ser auto-contidos, podem depender da disponibilidade de outros serviços ou da existência de um recurso, como por exemplo, de um banco de dados. Em um caso mais simples, um serviço pode calcular a raiz cúbica de um número fornecido sem depender de recurso externo, ou pode ter pré-carregado todos os dados necessários para sua vida útil. Por outro lado, um serviço que realiza a conversão de moeda preci- saria de acesso em tempo real a informações sobre taxa de câmbio para obter valores corretos (SRINIVASAN; TREADWELL, 2005).

[1] https://www.thoughtworks.com/insights/blog/microservices-nutshell

A figura 3 ilustra um simples ciclo de interação de serviços. O ciclo começa um serviço anunciando para um outro serviço de registro (1). Um cliente que pode ou não ser outro serviço, consulta o registro (2) para procurar um serviço que atenda às suas necessidades. O registro retorna uma lista de serviços adequados, o cliente seleciona um e passa uma mensagem de solicitação para ele, usando qualquer protocolo mutuamente reconhecido (3). Neste exemplo, o serviço responde (4) com o resultado da operação solicitada ou com uma mensagem de falha (SRINIVASAN; TREADWELL, 2005).

Os serviços normalmente são implementados com WS *(Web Services)*, fornecidos usando tecnologias como XML *(Extensible Markup Language)*, WSDL *(Web Services Description Language)*, SOAP *(Simple Object Access Protocol)* e UDDI *(Universal Description Discovery and Integration)*. Estas tecnologias proporcionam interoperabilidade aos serviços e garantem que os clientes possam encontrar e usar os serviços necessários, independente da tecnologia utilizada no serviço. Portanto, XML é fundamental para os WSs; WSDL é para descrever as interfaces do WS; SOAP fornece um protocolo para transferência de mensagens; e o UDDI atua como um repositório de WSs disponíveis.

A SOA oferece melhores oportunidades para integração de aplicativos corporativos com as vantagens adicionais de redução de custos, facilidade de manutenção e melhoria na flexibilidade e escalabilidade (MAHMOOD, 2007). Permite que as empresas e seus sistemas de TI sejam mais ágeis para as mudanças nos negócios e no ambiente (MAHMOOD, 2007).

Em larga escala, SOA oferece oportunidade para alcançar interoperabilidade e flexibilidade para se adaptar às mudanças tecnológicas. Desde que seja implementada corretamente, suas características técnicas se traduzem em benefícios, sendo eles:

*Conectividade e interoperabilidade*. Os serviços em uma arquitetura SOA utilizam o padrão XML. Ele se concentra na conversão de dados gerados por um componente específico no formato XML e a passa para o outro componente. A linguagem na qual os dados são gerados não é considerada (BOKHARI et al.,2015).

*Reutilização de sistemas existentes*. É possível integrar novos sistemas aos

sistemas legados, sem a necessidade de reescrever um novo sistema. Poupando despesas de capital, bem como tempo. Assim, as organizações não terão nenhuma sobrecarga de desenvolvimento de novos sistemas a partir do núcleo (BOKHARI et al., 2015).

*Alinhamento de TI em torno das necessidades do negócio*. Resulta em TI como a tecnologia que fornece valor agregado às operações de negócios (MAHMOOD,2007).

Embora as implementações de SOA demostrem ser uma solução para os requisitos de determinadas empresas, demandam alto custo e podem aumentar exponencialmente a complexidade e o esforço requerido para manter a aplicação. Os desafios da implemen- tação de estratégias SOA foram objeto de estudo na academia, a exemplo do que foi discutido em (PAPAZOGLOU et al., 2007) (HUTCHINSON et al., 2007) (MAHMOOD, 2007) (HUHNS; SINGH, 2005).

Como apresentado por (MAHMOOD, 2007), SOA demonstra não ser adequado nos seguintes cenários:

- Quando um aplicativo é uma entidade autônoma, em que não é necessário integrar entre componentes diferentes; um aplicativo que não é distribuído.

- Quando um aplicativo que não fornece funcionalidade completa ou não funciona como um sistema completo, em vez disso, serve como um componente e tem escopo limitado.

A arquitetura de microserviços será discutida na seção 2.4, cujo objetivo é eliminar os níveis desnecessários de complexidade como:

- *Segurança*. A interoperabilidade entre sistemas tornam estes mais expostos e vul- neráveis à ameaças externas. Portanto, é necessário maior controle ao nível de segurança;

- *Rastreabilidade*. Com a capacidade de orquestrar processos de negócio complexos, de igual forma aumentam a necessidade de monitoramento e rastreabilidade;

- *Gestão de Serviços de Metadados*. Os serviços em um ambiente baseado

em SOA exigem a troca de um grande número de mensagens para completar as tarefas. O gerenciamento das interações de tais serviços é uma tarefa complicada.

Desta forma, será possível se concentrar na programação de serviços simples que implementam efetivamente uma única funcionalidade. Microserviços são a segunda iteração no conceito de SOA e Computação Orientada a Serviços (DRAGONI et al., 2017).

## 2.3 OPORTUNIDADES PROPORCIONADAS PELA COMPUTAÇÃO EM NUVEM

Diante das oportunidades que a computação em nuvem proporciona, em essência, elas podem ser dividas em duas partes. Sendo a primeira parte operacional e os custos, como segunda parte.

Em mais detalhes, do ponto de vista operacional, a computação em nuvem oferece provisionamento rápido e automação de tarefas por meio de APIs, que permitem implantar e remover recursos de maneira instantânea. O que reduz o tempo de espera para o provisionamento de ambientes de desenvolvimento e homologação (testes); gerando aumento de produtividade.

Do ponto de vista econômico, o modelo *pay-per-use* não necessita de alto investimento inicial para adquirir recursos de TI ou para mantê-los, portanto, somente são pagos os recursos que efetivamente forem utilizados. Com isto, reduz a necessidade de manter uma infraestrutura física de TI e as empresas podem evitar *capex* (Despesas que uma empresa gasta para comprar, manter ou melhorar seus ativos fixos) em favor de *opex* (Capital utilizado para manter ou melhorar os bens físicos de uma empresa) e podem se concentrar no desenvolvimento ao invés do suporte às operações (TOFFETTI et al., 2015).

*Gerenciamento Facilitado*. A manutenção da infra-estrutura, seja de hardware ou software, é simplificada para a equipe de TI. Aplicações com grande volume de armazenamento são mais fáceis de usar no ambiente de nuvem comparado ao mesmo, quando gerenciado por conta própria. No nível de usuário, basta um navegador da Web com conectividade a internet (JADEJA; MODI, 2012).

*Gestão de Desastres*. Em caso de desastres, um backup externo é sempre útil.

Manter dados cruciais armazenados em backup, usando serviços de armazenamento em nuvem é a necessidade da hora para a maioria das organizações. Além disso, os serviços de armazenamento em nuvem não apenas mantém seus dados fora do local, mas também asseguram que eles tenham sistemas em vigor para a recuperação de desastres (JADEJA; MODI,2012).

Um dos principais benefícios das aplicações baseados na computação nuvem é a disponibilidade de forma contínua e onipresente para os usuários e seus colaboradores. A computação em nuvem facilita a quebra dos limites de um paradigma *single-workstation*.

## 2.4 ARQUITETURA BASEADA EM MICROSERVIÇOS

Arquitetura baseada em microserviços é uma solução promissora para lidar com os problemas relacionados a aplicativos monolíticos (FOWLER; LEWIS,2014). Eles surgiram da Arquitetura Orientada a Serviços (SOA) (NEWMAN, 2015; ZDUN; NAVARRO; LEYMANN, 2017) como um estilo de arquitetura para dividir aplicativos distribuídos em pequenos serviços independentemente implantáveis, cada um rodando em seu próprio processo e se comunicando via mecanismos leves (PAHL; JAMSHIDI,2016). Devido ao serviço ter tamanho reduzido e foco em apenas uma área de negócio, os microserviços possibilitam alcançar uma boa modularidade na base de código.

O termo "micro"não é tão importante e não está diretamente relacionado ao tamanho, mas sobre ter esses serviços separados, não importando se são dois, três ou centenas, talvez se estiver trabalhando com cem serviços, as coisas vão se tornar muito mais difíceis. É muito importante considerar apenas a idéia de que talvez o sistema seja melhor para sua aplicação como um conjunto de serviços menores.

A Figura 10 ilustra uma aplicação *frontend* que se comunica com seis serviços diferentes. Os serviços *OrderingService* e *PaymentService* separadamente, processam a criação do pedido e pagamento; estes serviços têm suas bases de código separadas e a comunicação entre eles é feita através de APIs fornecidos por esses serviços.

Quando os limites dos serviços estão bem definidos, as chances de estarem em conformidade com o Princípio de Responsabilidade Única (SRP) (NEWMAN, 2015) aumentam. Este princípio diz: "Reunir as coisas que mudam, pelas mesmas razões. Separar as coisas que mudam por razões diferentes." (MARTIN, 2014).

O SRP é um dos princípios do SOLID, que são os cinco princípios básicos de um bom design de software orientado a objetos. A arquitetura de microserviços facilita o SRP, pois o estilo desta arquitetura incentiva a criação de pequenas unidades. Portanto, o objetivo é ter serviços que sejam fracamente acoplados e altamente coesos (NEWMAN, 2015). Baixo acoplamento, alta coesão, SRP e modularidade são considerados boas práticas e auxiliam na melhoria da arquitetura de software. É comum que quando o tamanho da organização e o tamanho da base de código aumentam, fica mais difícil manter essa qualidade no software (RICHARDSON, 2015). A arquitetura de microserviços promove algumas vantagens.

*Ciclo de Desenvolvimento Acelerado*. Quando os limites estão bem definidos e cada serviço responde por uma determinada área de negócio da aplicação é provável que os ciclos de desenvolvimento possam ser acelerados. Habilitando aos desenvolvedores a capacidade de mais rapidamente entregar modificações (NEWMAN, 2015).

Uma arquitetura de microserviços pode fazer melhor uso da elasticidade e modelo de preços de ambientes de nuvem (BALALAIE; HEYDARNOORI; JAMSHIDI,2016). Além disso, eles fornecem a flexibilidade para migrar e realizar a interoperabilidade entre diferentes infraestruturas e plataformas de nuvem. De fato, da perspectiva do usuário, a interação com o sistema permanece a mesma. O ponto principal é que, como já mencionado, o estilo arquitetural de microserviços traz inúmeros benefícios quando comparado a uma arquitetura monolítico (KWAN et al.,2016).

O estilo arquitetural de microserviços, foi sugerido como uma solução para as deficiências da arquitetura monolítica pelas seguintes razões: (1) reduzem a complexidade usando serviços de pequena escala; (2) eles escalam e implantam facilmente partes do sistema; (3) melhoram a flexibilidade para usar diferentes estruturas, ferramentas e tecnologias; (4) aumentam a escalabilidade geral; e (5) melhoram a resiliência do sistema (AMARAL et al.,2015).

Em seguida, segue uma lista não exaustiva de vantagens que se destacam quando se usam microserviços: serviços coesos e pouco acoplados (WOLFF,2016); implementação independente e portanto, adaptabilidade (MILLETT, 2015); independência de equipes multifuncionais, autônomas e organizadas para fornecer valor comercial, não apenas carac- terísticas técnicas (MILLETT, 2015); independência de conceitos de domínio (WOLFF, 2016); livre de efeitos colaterais potenciais (SPoF) em serviços; incentiva a cultura de DevOps (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016), que representa basicamente a ideia de descentralizar a concentração de habilidades em equipes multifuncionais, enfati- zando a colaboração entre desenvolvedores e equipes, o que garante redução do tempo de entrega e maior agilidade durante o desenvolvimento do software.

## 2.5 CONTAINERS

*Container* é um ambiente isolado que contém todo o código necessário para executar um serviço em um sistema operacional. Suas dependências e configuração são abstraídas através de arquivos manifesto de implantação; e são empacotados juntos como uma imagem (WOLFF, 2016). Eles fornecem uma solução para a portabilidade de aplicações, pois não há vínculo com máquinas físicas. Uma aplicação que utiliza container pode ser testada como uma unidade isolada e implementada como uma instância de imagem do *container* no sistema operacional do host. O Docker[2] é um exemplo de ferramenta para conteinerização de aplicações.

Os *container's* não foram criados para microserviços. Eles surgiram como resposta a uma necessidade prática: as equipes de tecnologia precisavam de um conjunto de ferra- mentas capazes de implantar aplicativos complexos de maneira universal e previsível. Ao empacotar um aplicativo como um contêiner Docker, que pressupõe o pré-agrupamento de todas as dependências necessárias nos números de versões corretas, é possível a implantação de forma confiável em qualquer instalação de hospedagem na nuvem ou local, sem se preocupar com o ambiente de destino e a compatibilidade (NADAREISHVILI et al.,2016). Com base na documentação do Docker é possível afirmar que ele impulsiona a adoção de microserviços. Umas das

[2] https://www.docker.com

razões para esta afirmação é que o Docker dá ênfase nos containers à "filosofia unix", que preconiza "faça uma coisa e faça bem". Este prinícipio é destacado de forma proeminente na própria documentação do Docker: *"Run only one process per container. In almost all cases, you should only run a single process in a single container. Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers."* [3]

## 2.6 DEVOPS

Em essência, DevOps é uma metodologia que aproxima o desenvolvimento (Dev) das operações (Ops) em um único processo integrado e contínuo. Estimulando pessoas, processos e tecnologia para uma maior colaboração e inovação em todo o processo de desenvolvimento e lançamento de software. Para a adoção desta metodologia necessita- se que sejam realizadas mudanças a nível organizacional, pois é comum ter times de desenvolvimento e operação trabalhando em equipes distintas (WOLFF, 2016).

As habilidades técnicas se entrelaçam, as operações funcionam mais sobre automação e testes, o que acaba no final, fazendo parte do processo de desenvolvimento de software. Ao mesmo tempo, o monitoramento, a análise de logs e a implantação também se tornam cada vez mais tópicos para desenvolvedores.

DevOps e Microserviços trabalham muito bem juntos. A ideia que as equipes implantem microserviços até a produção e continuem cuidando deles, só pode ser alcançado com equipes de DevOps. Essa é a única maneira de garantir que as equipes tenham o conhe- cimento necessário sobre todo o processo (WOLFF, 2016) (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016).

## 2.7 CONCLUSÃO DO CAPÍTULO

Este capítulo discorreu acerca do referencial teórico necessário para a compreensão desta dissertação. Apresentou as características e limitações da arquitetura monolítica, as dificuldades e desvantagens encontradas na arquitetura orientada a

[3] https://docs.docker.com/develop/develop-images/dockerfile_best − practices

` serviços, as oportunidades proporcionadas pela computação em nuvem, arquitetura de microserviços, containers e devops.

# 3 ESTUDO PILOTO

Este capítulo descreve o projeto e as configurações de um estudo exploratório em duas fases, com o objetivo de identificar etapas relevantes e eficazes de um roteiro preliminar, para apoiar a migração de sistemas de software legados para uma arquitetura baseada em microserviços. Pelo fato de ser um estudo piloto, tem-se que o roteiro proposto ainda será uma primeira versão a partir da qual serão identificadas oportunidades de melhoria.

Os estudos exploratórios têm a intenção de estabelecer as bases para futuros trabalhos empíricos (SEAMAN, 1999). A identificação de um roteiro eficaz é fundamental para enfrentar desafios desse tipo.

## 3.1 CONTEXTO

O estudo piloto considerou um exemplo típico de sistema Web, chamado *ePromo*, para emissão de cupons de desconto. Este sistema foi implementado utilizando a linguagem de programação PHP para gerenciar campanhas de divulgação. Os principais recursos disponíveis, incluem: criação de ofertas personalizadas e emissão de tickets feitos pelo cliente. Todas estas funcionalidades são implementadas em um grande artefato, que roda em um servidor web *Nginx*, conectado a um único banco de dados relacional (MySQL). Utiliza o Memcached como um sistema de cache em memória, incluindo os dados relacionados às sessões; sinais de uma aplicação monolítica.

O cenário em que esta aplicação estava rodando era de crescimento repentino, o que causou um rápido aumento na demanda por geração de cupons. Devido ao aumento súbito, a aplicação começou a enfrentar problemas no componente responsável pela emissão dos cupons, o que levou à interrupções na operação do sistema.

## 3.2 PLANEJAMENTO DO PROJETO PILOTO

Este estudo piloto teve o objetivo de migrar um sistema de software legado para uma arquitetura de microserviços. A primeira ação executada foi identificar os artefatos do

sistema, baseando-se nas funcionalidades do sistema. Partindo dos detalhes obtidos durante a identificação dos artefatos, um mapa de contexto foi criado para garantir mais clareza do domínio. A partir do mapa de contexto foi possível identificar melhor os contextos e o nível de acoplamento, tanto no código-fonte quanto na estrutura do sistema.

Em seguida, isolar em camadas bem definidas e que expressam as camadas de domínio, lógica de negócios, infraestrutura e interface do usuário. Por fim, o objetivo foi obter mais detalhes do domínio, utilizando o conceito de contexto delimitado do DDD para mostrar a viabilidade na conversão dos contextos identificados para uma arquitetura de microserviços. E então, coletar as referências aprendidas durante este estudo piloto para aplicar ao estudo de caso. Por fim, responder as seguintes questões de pesquisa:

a) Questão de Pesquisa 1 (QP1): Como encontrar funcionalidades em um sistema de software legado para posteriormente serem modularizados e adaptadas à uma arquitetura baseada em microserviços?

b) Questão de Pesquisa 2 (QP2): Como migrar funcionalidades candidatas para uma arquitetura baseada em microserviços?

c) Questão de Pesquisa 3 (QP3): Quais as etapas mínimas necessárias para compor um guia de migração de sistemas de software legados, para uma arquitetura baseada em microserviços?

## 3.3 CONCEITOS UTILIZADOS NO ESTUDO PILOTO

***Os Sistemas Legados***. A aplicação candidata para este estudo deve possuir as seguintes características: (1) ser uma aplicação legada, (2) ter uma arquitetura monolítica que não tenha suas funcionalidades modularizadas, (3) apresentar sintomas de espalhamento e emaranhamento e (4) possuir as características descritas pelo anti-padrão *Big Ball Of Mud* (COPLIEN; SCHMIDT,1995).

***O que se Espera.*** A partir da versão evoluída, é esperado que aplicação seja coerente, tenha baixo acoplamento, com uma decomposição modular mais alinhada

com os serviços que presta (NEWMAN,2015). Também é esperado observar um aumento na auto- nomia de desenvolvimento de equipes dentro da organização, já que novas funcionalidades podem ser localizadas dentro de serviços específicos (NEWMAN, 2015).

***Conceitos Chave DDD***. Para realizar as tarefas deste estudo, foram utilizados os principais conceitos de DDD para traduzir as funcionalidades para o domínio e subdomínio e assim, suportar a migração, são eles: *Bounded Context*, *Domain Model*, *Ubiquitous Language* e *Context Map*.

O *Bounded Context* é um subsistema no espaço da solução, com limites claros que o distinguem de outros subsistemas (EVANS,2004). *Bounded Context* ajuda na separação de contextos para entender e abordar complexidades com base em intenções comerciais. O *Domain*, no sentido amplo da palavra, é todo o conhecimento em torno do problema que se está tentando resolver. Portanto, ele pode se referir a todo o *Business Domain* ou apenas à uma área básica ou de suporte. Em um *Domain*, espera-se transformar um conceito técnico com um modelo (*Domain Model*) em algo compreensível.

O *Domain Model* é o conhecimento organizado e estruturado do problema. Este modelo deve representar o vocabulário e os principais conceitos do problema de domínio e identificar as relações entre todas as entidades. Deve funcionar como uma ferramenta de comunicação para todos os envolvidos, criando um conceito muito importante no DDD, que é *Ubiquitous Language*. Este modelo pode ser um diagrama, exemplos de código ou até mesmo documentação escrita do problema. O importante é que o *Domain Model* deve ser acessível e compreensível por todos os envolvidos no projeto. Um *Context Map* é um diagrama de alto nível, que mostra uma coleção de contextos delimitados e os relacionamentos entre eles (EVANS, 2004).

O *Aggregate Root* tem como objetivo selecionar o objeto que é a "raiz"de um grupo de objetos, de uma maneira abstrata, de uma fachada para representar toda a estrutura. Por outro lado, o *Value Object* pode incluir valores simples ou compostos que tenham um significado comercial.

## 3.4 EXECUÇÃO DE ATIVIDADES DO ESTUDO PILOTO

Para responder **QPD1** (Como encontrar funcionalidades em um sistema de software legado para posteriormente serem modularizados e adaptadas à uma arquitetura baseada em microserviços?) foi executado um processo iterativo de identificação manual de recursos candidatos e seus respectivos relacionamentos, navegando entre os diretórios e arquivos, identificando os propósitos e responsabilidades de cada um dos artefatos. A figura 4 ilustra as entidades identificadas: Offer, OfferPoint, Ticket, Requirement, Timer, User, Company no início do estudo piloto.

Figura 4 – Entidades e Funcionalidades Dispersas e Emaranhadas no Sistema ePromo

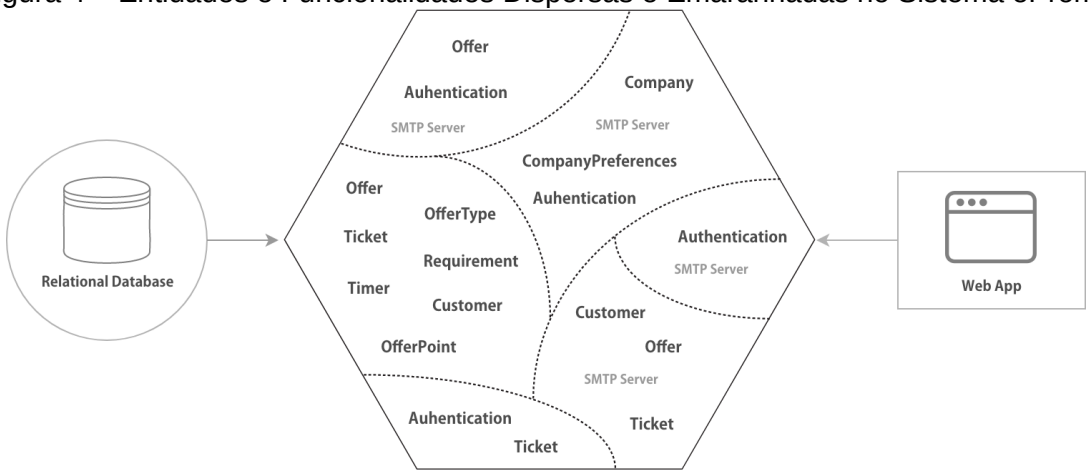
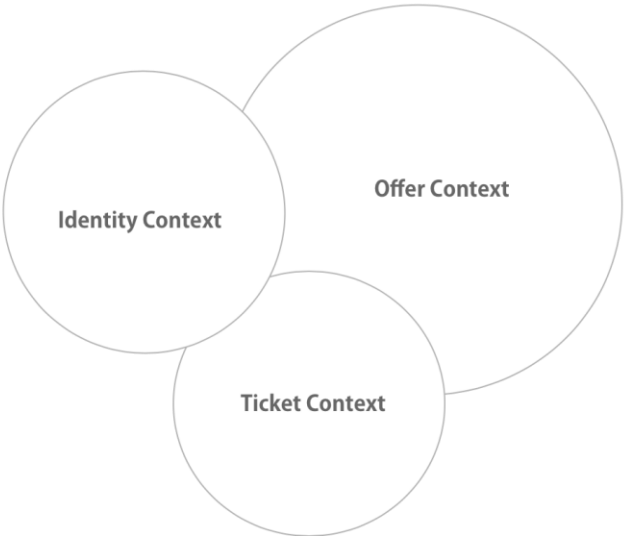



Figura 5 – Mapa de Contexto do Sistema ePromo



Ao analisar os recursos associados à essas entidades, adquirimos uma percepção inicial de como esses artefatos estão emaranhados e dispersos no código. As funcionalidades, são as principais referências para construir o mapa de contexto, como representado na figura 5. Vale ressaltar que, durante a elaboração do mapa de contexto com base nas informações recuperadas do código-fonte, foi possível reconhecer as entidades, objetos de valor e agregados. Durante a elaboração do mapa de contexto, é possível identificar um código acoplado à estrutura da aplicação Web, logo no estágio de navegação inicial. Os seguintes candidatos a objetos de valor foram identificados: OfferType, AmountOff, PercentOff, FreeStuff, Customer, TimeInterval.

É importante lidar com um recurso de cada vez, com base na lista de recursos elaborada durante a fase 1. Nesta estratégia, adotou-se inciar pela funcionalidade que teria o menor impacto quando comparada às outras. Isso permitiria a validação dos limites entre os recursos com menor risco de efeitos colaterais. Considerando que as regras de negócio estavam dispersas nos controladores com significativa duplicação, era necessário um esforço adicional para identificar as várias funcionalidades envolvidas. Este cenário também indicou um sintoma de emaranhamento.

Ao analisar o artefato TicketsController, verificou-se que ele tem muitas responsabilidades e que suas regras de negócios estavam dispersas. Precisava de refatoração extensiva, incluindo extração de camadas mais bem definidas para diferentes níveis de abstração. Sendo que cada camada estaria representada por um diretório, que implicou mudanças estruturais nesse nível, na raiz de origem do repositório.

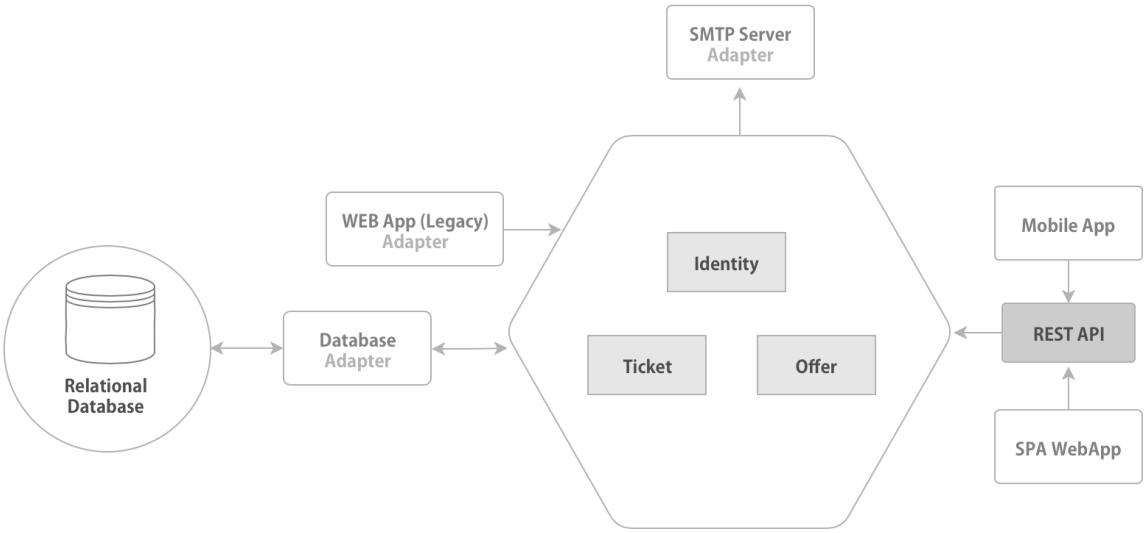
Foram criados novos diretórios: Application, Domain e Infrastructure. O diretório Application não deve ter qualquer lógica de negócios, deve ser responsável por conectar a interface do usuário às camadas inferiores, ou seja, a camada de aplicação será capaz de se comunicar com a camada de domínio que funcionará como uma espécie de API pública para o aplicativo. Aceitará pedidos do mundo exterior e retornará respostas apropriadamente.

O diretório Domain irá conter todos os conceitos, regras e lógica de negócios do

sistema, como por exemplo, a entidade ou repositório do usuário. Esses arquivos serão armazenados de acordo com o contexto identificado nas etapas anteriores. O diretório Infrastructure irá hospedar as implementações referentes a recursos técnicos, que forne- cem suporte às camadas acima. Por exemplo, a camada de persistência e comunicação através de redes.

Neste ponto foi aplicado o padrão *Command* (GAMMA,1995) para minimizar o acoplamento e lidar com o código emaranhado, com as regras de negócios que estavam espalhadas nos controladores do sistema. O padrão *Command* representa a intenção do usuário, alguma tarefa a ser executada. É um simples *Data Transfer Object* (DTO) que contém apenas valores de tipos primitivos. Cada comando terá um manipulador, que é um serviço da aplicação, que sabe lidar com um comando que lhe foi atribuído. O manipulador de comando usa os dados do objeto de comando para criar um agregado ou buscar em um repositório e executar alguma ação (GAMMA,1995). O papel do controlador é encaminhar apenas as informações necessárias para criar o comando, neste caso o CreatingTicket, para então encaminhar ao manipulador, que lidará com a aceitação do comando até concluir sua tarefa.

Figura 6 – Versão Modularizada do ePromo (Final do Estudo Piloto)




Essa abordagem traz várias vantagens, a saber: (1) a funcionalidade pode ser executada em qualquer parte do aplicativo; (2) o controlador não terá mais regras de

negócio, fazendo exatamente o que foi proposto acima; (3) os testes podem ser facilitados, como resultado do desacoplamento. A nova versão do sistema modularizado é apresentada na figura 6.

## 3.5 COLETA E ANÁLISE DOS DADOS

Para coletar os dados necessários para obtenção das respostas das três questões de pesquisa elaboradas no estudo exploratório, foi necessário aplicar o processo de identificação manual dos recursos candidatos e seus respectivos relacionamentos, navegando entre os diretórios e arquivos, identificando os propósitos e as responsabilidades de cada um dos artefatos. Como pode ser observado na figura 4 as funcionalidades estavam dispersas e emaranhadas, foi possível a partir da coleta e análise destes dados, identificar diferentes conceitos e responsabilidades como: Entidades e Objetos de Valor.

## 3.6 RESULTADOS

Como resultado da execução das atividades previstas para a migração neste estudo piloto, não foi possível migrar para uma arquitetura de microserviços, devido ao cenário relativamente simples, percebido através do mapa de contextos. Nestas condições, foi indicado apenas executar a primeira parte do roteiro preliminar, relacionado à reestrutu- ração do sistema legado para uma versão modularizada. Portanto, foi possível somente atender ao OE2 (Analisar até que ponto um conjunto de etapas apoiaria na busca de funcionalidades candidatas que poderiam ser moduladas no contexto de aplicações legadas para posteriormente serem convertidas em microserviços).

## 3.7 LIÇÕES APRENDIDAS DO ESTUDO PILOTO

Como lição aprendida deste estudo piloto, verificou-se que devido a complexidade introduzida pela arquitetura de microserviços, é necessário ter ciência do esforço ao decidir pela migração para trabalhar na implantação automatizada dos serviços, monitoramento, tratamento de falhas, consistência eventual, entre outros fatores.

Por essas razões foi decidido não optar pela migração e manter o sistema *ePromo* em sua nova versão modularizada.

Não há um relação custo-benefício positivo entre as vantagens dos microserviços e os custos e esforços correspondentes necessários para gerenciá-lo (SINGLETON, 2016). Embora as abordagens de microserviços ofereçam benefícios substanciais; a arquitetura correspondente requer maior poder de processamento e gerenciamento, o que também pode impor aumentos substanciais de custos (SINGLETON, 2016). Isso poderia acarretar no aumento da complexidade, o que é incompatível com o cenário.

## 3.8 CONCLUSÃO DO CAPÍTULO

Este capítulo descreveu o sistema ePromo e as motivações que levaram à aplicação da migração. Explorou as etapas da migração do sistema, com o apoio das fases sugeridas no guia proposto. Por fim, apresentou as lições aprendidas, baseado na experiência adquirida durante a aplicação do estudo piloto.

# 4 ESTUDO DE CASO

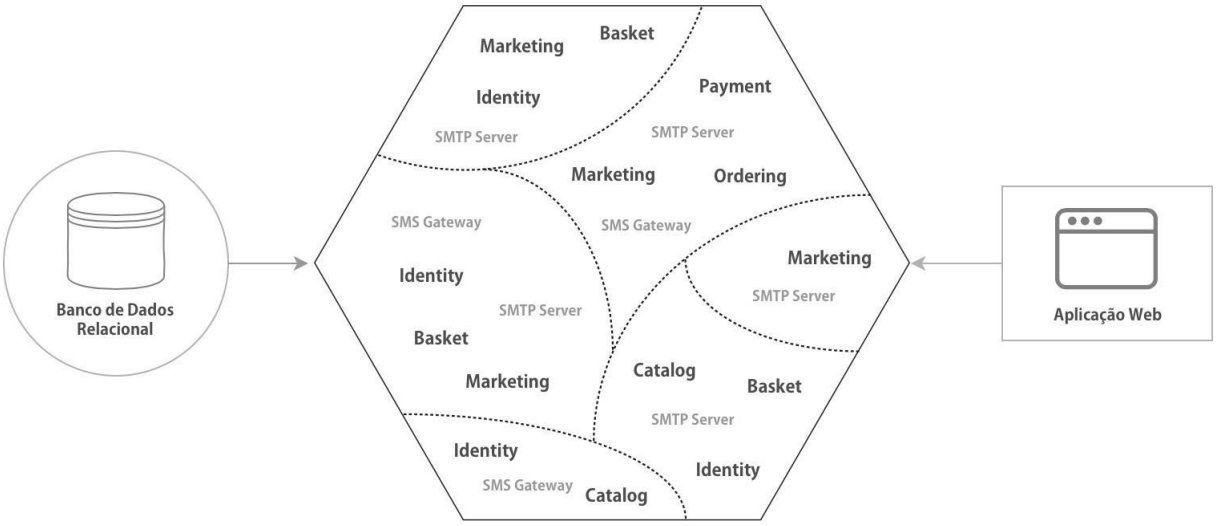
Este capítulo busca uma versão revisada e aprimorada do roteiro preliminar, proposto no capítulo anterior, através de um estudo de caso da migração de um sistema de software monolítico legado de loja online para uma arquitetura baseada em microserviços.

## 4.1 CONTEXTO

O estudo de caso considerou um sistema típico de loja online *eShop*, como ilustrado na Figura 7. Este sistema fornece aos usuários um catálogo online de produtos. Dentre as principais funcionalidades que integram este sistema, podem ser citadas: autenticação do usuário, catálogo de produtos, ofertas especiais e pagamentos. Todas as funcionalidades são implementadas utilizando a linguagem de programação PHP, dentro de um grande módulo que é conectado a um banco de dados relacional (MySQL). A característica monolítica ocorre em função do sistema ser executado como um único artefato em servidor web. Este sistema roda um servidor web *Nginx*.

Ao longo dos anos, o tamanho do código-fonte do sistema aumentou drasticamente, conforme as partes interessadas solicitavam novas mudanças e funcionalidades. Para lidar com essas solicitações, os desenvolvedores se esforçaram para fornecer novos lançamentos, o que exigiu mais esforços.

Figura 7 – Um Sistema Tradicional de Software Legado Monolítico (Estudo de Caso)



## 4.2 PLANEJAMENTO DO ESTUDO DE CASO

Este estudo de caso teve o objetivo de migrar um sistema de software legado para arquitetura de microserviços, separando em duas etapas. A primeira etapa identificou os artefatos do sistema, baseado-se nas funcionalidades e responsabilidades do domínio. A partir dos detalhes obtidos durante a identificação dos artefatos, foi criado um mapa de contexto para garantir mais clareza do domínio. Com os contextos identificados, eles são isolados em camadas bem definidas que expressam o modelo de domínio, lógica de negócios, infraestrutura e interface do usuário.

A segunda parte obtém mais detalhes do domínio, utilizando o conceito de contexto delimitado do DDD para mostrar a viabilidade na conversão dos contextos identificados para uma arquitetura de microserviços. Em seguida, este estudo de caso responde as seguintes questões de pesquisa:

- Questão de Pesquisa 1 (QP1): Como encontrar funcionalidades em um sistema de software legado para posteriormente serem modularizados e adaptadas à uma arquitetura baseada em microserviços?

- Questão de Pesquisa 2 (QP2): Como migrar funcionalidades candidatas para uma arquitetura baseada em microserviços?

- Questão de Pesquisa 3 (QP3): Quais as etapas mínimas necessárias para compor um guia de migração de sistemas de software legados, para uma arquitetura baseada em microserviços?

## 4.3 EXECUÇÃO DAS ATIVIDADES DO ESTUDO DE CASO

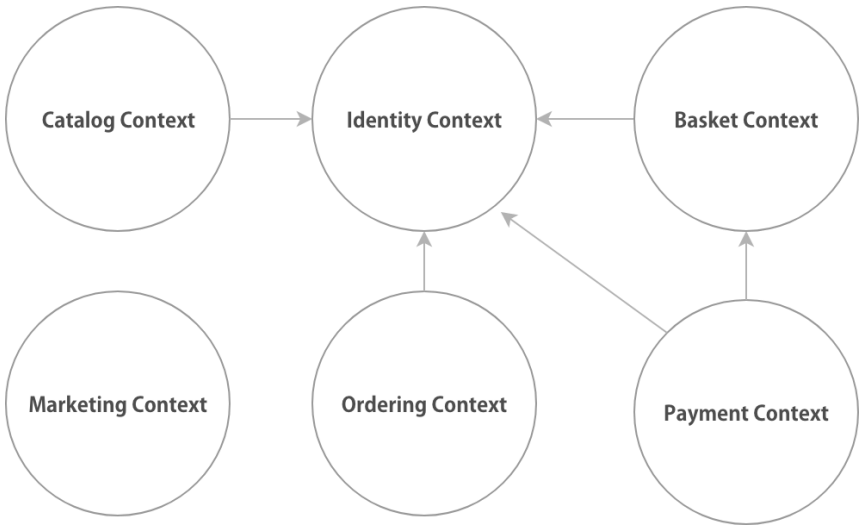
As atividades deste estudo de caso foram divididas em duas partes.

*Parte I - Migrando o Sistema com Características de Espalhamento e Entrelaçamento para uma Versão Modularizada*. As funcionalidades candidatas foram identificadas de forma manual, navegando entre os diretórios e arquivos para descobrir o objetivo de cada artefato, assim como foi feito no estudo piloto. O que

responde a **QPD1**.

A figura 7 ilustra as entidades: Identity, Basket, Marketing, Catalog, Ordering e Payment relacionadas às funcionalidades identificadas.

Figura 8 – Um Mapa de Contexto para o Sistema de Software Monolítico Legado (Estudo de Caso)




Este é o resultado do primeiro passo, que visa identificar as principais funcionalidades e responsabilidades, em vista de um estabelecimento provisório de limites entre elas. Em seguida, é planejado dividir o módulo principal em unidades. A chave para essa tarefa foi o uso de contextos delimitados e seus respectivos relacionamentos, conforme representado na figura8. Em cada contexto delimitado foram aplicados os seguintes conceitos-chave de DDD: *Aggregate Root*, *Entity*, *Value Object*, *Services*, *Repositories* e *Factories*.

Esses conceitos ajudam a gerenciar a complexidade do domínio e garantem mais clareza do comportamento no modelo de domínio. Depois de identificar os contextos, deve ser classificado o nível de complexidade, começando pelos mais simples, para validar o mapeamento de contexto.

Os contextos foram colocados em camadas bem definidas, expressando o modelo de domínio e a lógica de negócios, eliminando as dependências de infraestrutura, interfaces de usuário e lógica de aplicativos, que muitas vezes se misturam. Todo

código relacionado ao modelo de domínio deve ser concentrado em uma camada, isolando-o das camadas superiores, como a camada de interface do usuário, camada de aplicação e de infraestrutura (EVANS, 2004).

Em alguns casos o padrão *Strangler* (TAIBI; LENARDUZZI; PAHL, 2017) pode ser aplicado para lidar com a complexidade do módulo que será refatorado. Pois ele cria uma valor incremental em curto período, quando comparado a migrações tipo "big bang", que necessita da implementação em todo código-fonte antes de liberar qualquer nova funcionalidade.

Um diretório deve ser criado para cada um dos contextos delimitados e dentro de cada um dos diretórios, três novos diretórios devem ser adicionados, sendo um para cada camada: Domain, Application, Infrastructure. Eles devem conter o código-fonte necessário para que esse contexto delimitado funcione. É crucial considerar os modelos de domínio e seus invariantes e reconhecer *Entities*, *Value Object's* e também *Aggregates*. O código-fonte deve ser mantido nesses diretórios conforme descrito na sequência. O diretório Application deve conter todos os serviços de aplicação e manipuladores de comandos.

O diretório Domain contém as classes com padrões táticos existentes no DDD, como: Entity, Value Object, Domain Event, Repository e Factory. O diretório Infrastructure deve fornecer os recursos técnicos para outras partes do aplicativo, isolando toda a lógica de domínio dos detalhes da camada de infraestrutura.

Este último contém em mais detalhes, o código para enviar e-mails, enviar mensagens, armazenar informações no banco de dados, processar solicitações HTTP e fazer solicitações para outros servidores. Qualquer estrutura e biblioteca relacionada ao mundo externo, como sistemas de rede e de arquivos, deve ser usada ou chamada pela camada de infraestrutura.

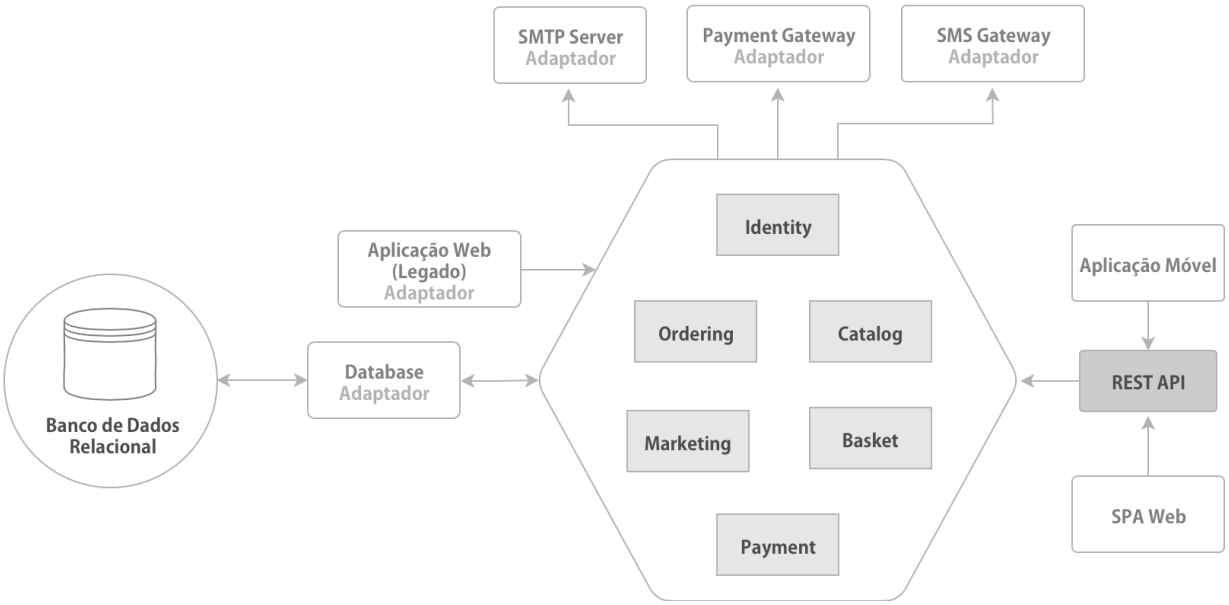
A estrutura de diretórios de um contexto delimitado ficaria organizado da seguinte maneira:

```
+--src
|           +-- eShop
|              +-- Marketing
|                 +-- Application
|                 +-- Domain
|                 +-- Infrastructure
+-- tests
```

A regra de dependência (MARTIN, 2017), afirma que você deve confiar apenas em coisas que estão na mesma camada ou em uma camada mais profunda. Assim, o código do domínio só pode depender de si mesmo, o código do aplicativo deve depender apenas do código de domínio e o código de infraestrutura pode depender de qualquer uma das camadas. O código de domínio também não pode depender do código da infraestrutura.

*Parte II - Migrando da Versão Modularizada para microserviços* Neste ponto, o foco é a análise do mapa de contexto previamente desenvolvido e a avaliação da viabilidade de decompor cada contexto identificado em candidatos a microserviços. Durante a análise do mapa de contexto é necessário entender e identificar as relações e dependências organizacionais. Isso é análogo à modelagem de domínio, que pode começar de modo relativamente superficial e aumentar gradualmente os níveis de detalhes.

Figura 9 – Um Sistema Legado Monolítico Evoluído (Estudo de Caso)




A maneira mais comum usada para decompor um sistema em partes menores é baseada na segmentação em camadas com base na interface do usuário, na lógica de negócios e nas responsabilidades do banco de dados. No entanto, esta maneira está mais propícia à geração de um alto acoplamento entre os módulos, causando a replicação da lógica de negócios nas camadas do aplicativo (DRAGONI et al., 2017); acoplamento define o grau de dependência entre componentes ou módulos de um aplicativo.

A proposta de microserviços para contornar esse problema implica segmentar o sistema em partes menores, com menos responsabilidades. Além disso, também considera os contextos de domínio e aplicação, gerando um conjunto de serviços autônomos com acoplamento reduzido.

O *Domain-Driven Design* (DDD) (EVANS, 2004) fornece conceitos-chave para apoiar a decomposição de um sistema de software em microserviços, que responde ao OE3. O DDD provê ferramentas para representar o mundo real na arquitetura, por exemplo, usando uma das ferramentas da parte estratégica, contextos delimitados para representar unidades organizacionais que identificam e focalizam o domínio principal. Essas características levam a uma melhoria na qualidade da arquitetura de software (EVANS, 2004).

Para responder **QPD2**, o padrão Contexto Delimitado do DDD é utilizado para

organizar e identificar os microserviços (NADAREISHVILI et al., 2016). Muitos defensores da arquitetura de microserviços usam a abordagem DDD de Eric Evans, pois oferece um conjunto de conceitos e técnicas que suportam a modularização em sistemas de software. Evans defendeu os contextos delimitados como facilitadores da criação de componentes menores e mais coerentes (modelos), que não devem ser compartilhados entre contextos.

No mapa de contexto mostrado na figura8, a seta foi utilizada para facilitar a identificação dos relacionamentos *upstream/downstream* entre os contextos. Quando um contexto delimitado tem influência sobre outro (devido a fatores de natureza menos técnica), fornecendo algum serviço ou informação, essa relação é considerada *upstream*. No entanto, os contextos delimitados que o consomem, compreendem como um relacionamento *downstream* (EVANS,2004).

A correta identificação de contextos delimitados usando DDD e a quebra de um sistema grande entre eles é uma forma eficaz de definir limites de microserviços, como aponta (NEWMAN,2015): *"Se nossos limites de serviço se alinharem aos contextos delimi- tados em nosso domínio e nossos microserviços representarem esses contextos delimitados, teremos um excelente começo para garantir que nossos microserviços sejam fracamente acoplados e fortemente coesos"*.

Newman também aponta que os contextos delimitados representam domínios de negócios autônomos (ou seja, recursos de negócios distintos) e, portanto, são o ponto de partida apropriado para identificar limites para microserviços.

O uso de DDD e contextos delimitados reduz as chances de dois microserviços precisarem compartilhar um modelo e o espaço de dados correspondente, arriscando um acoplamento rígido. Evitar o compartilhamento de dados facilita o tratamento de cada microserviço como uma unidade de implantação independente. A implantação independente aumenta a velocidade, mantendo a segurança dentro do sistema. DDD e Contextos Delimitados parecem fazer um bom processo para projetar componentes (NEWMAN, 2015). Note, no entanto, que é possível usar DDD e ainda assim acabar com componentes bastante grandes, algo que contraria os princípios da arquitetura de microserviços. Em suma, menor é melhor.

Considerando que um sistema legado geralmente tem vários contextos delimitados,

não é uma tarefa trivial decidir a partir de qual contexto deve-se iniciar a migração para microserviços. A seleção de contextos que não possuem nenhum ou o menor nú- mero de relacionamentos, pode ser uma estratégia eficiente para iniciar a migração para microserviços. Na seqüência, outros contextos identificados podem ser avaliados como adequados sempre que ordenados pelo número de relacionamentos. Com base no mapa de contexto apresentado na figura8, fica fácil descobrir que o MarketingContext não possui qualquer relação com outros contextos. Por esse motivo, este contexto é um bom candidato a conversão para um microserviço e pode ser usado como ponto de partida do processo. O próximo contexto migrado para microserviços foi o CatalogContext, seguido de BasketContext, PaymentContext e IdentityContext.

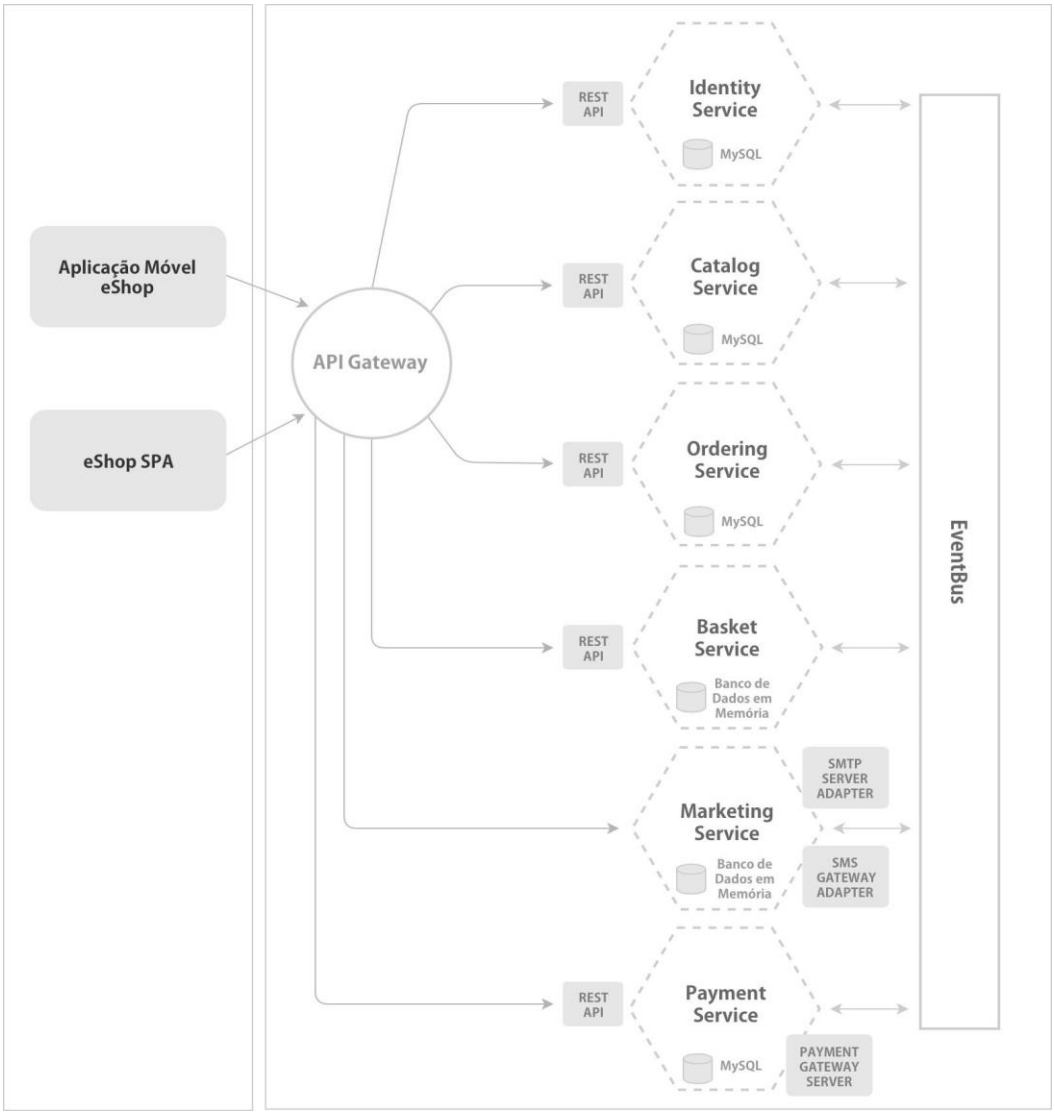
As características utilizadas para selecionar os contextos a serem migrados para microserviços, foram: número de relacionamentos entre outros contextos; o nível de com- plexidade de cada contexto e a relevância do contexto para o domínio em análise. Cada serviço deve possuir sua própria estrutura, permitindo assim, a manutenção separada de repositórios externos para cada microserviço. Isso facilita a evolução e os ajustes da implementação, que podem ser realizados em serviços específicos, evitando possíveis efeitos colaterais (SPoF) em outros serviços.

Uma característica importante de um serviço é seu baixo número de responsabilida- des, que é reforçado pela definição do *Single Responsibility Principle* (SRP) (MARTIN, 2002). Cada serviço deve ter um limite bem definido entre os módulos, que deve ser criado e publicado de forma independente, por meio de um processo de implantação automatizado. Microserviços são serviços autônomos e pequenos que trabalham juntos (NEWMAN, 2015). Uma equipe pode trabalhar em um ou vários contextos delimitados, cada um servindo de base para um ou vários microserviços. Mudanças e novos recursos devem estar relacionados a apenas um contexto delimitado e, portanto, apenas uma equipe (WOLFF, 2016).

A arquitetura baseada na separação de microserviços facilita o uso de outros padrões já reconhecidos na literatura. Por exemplo, o padrão *publish-subscribe* implementado na forma de EventBus e Domain Events, permite e simplifica a comunicação entre os microserviços que compõem o aplicativo. Um uso dessa abordagem pode ser visto em dois serviços (Ordering e Payment) que estabelecem

comunicação durante a captura do mesmo evento, embora residam em ambientes diferentes. A adoção de uma arquitetura auxiliar como EventBus e Domain Events proporciona maior isolamento na comunicação entre os serviços, mas é importante salientar que esta abordagem também pode aumentar a complexidade no gerenciamento da consistência em caso de falha.

Figura 10 – Um Novo Sistema de Software Baseado na Arquitetura de microserviços (Estudo de Caso)




Manter todos os dados em uma única base é contrário às características de gerenciamento de dados descentralizado dos microserviços. A estratégia é mover os recursos verticalmente, desacoplando o recurso principal em seus dados e redirecionando todos os aplicativos front-end para as novas APIs.

Ter vários aplicativos usando os dados de um banco de dados centralizado é o

principal bloqueio para desacoplar os dados juntamente com o serviço. Portanto, é necessário incorporar uma estratégia de migração de dados apropriada ao seu ambiente, independentemente de ser capaz de redirecionar e migrar todos os dados de *read/write* ao mesmo tempo.

Migrar dados de um sistema existente é um processo complexo. Requer cuidados especiais que dependem da situação específica. Foi decidido realizar a migração do banco de dados do sistema *eShop* em pequenos etapas. Foram selecionadas as tabelas relacionadas a cada serviço e em seguida criado um novo esquema de banco de dados (MySQL) para o respectivo serviço. Depois, migrar um por um. O banco de dados não era particularmente grande e essa abordagem foi aplicada sem efeitos colaterais. No entanto, essa abordagem pode não ser a mais eficiente, dependendo do tamanho do banco de dados a ser migrado.

Cada cenário deve ser analisado de maneira específica e em mais detalhes. A biblioteca *Doctrine Migrations*[4] [1] foi utilizada para apoiar o processo de migração.

Baseado no roteiro proposto, a figura 10 mostra a arquitetura do novo sistema baseado em microserviços.

## 4.4 RESULTADOS

Como resultado da experiência adquirida no estudo das duas fases relatadas anteriormente, foram identificados quatro desafios principais no processo de migração. A descrição desses desafios atendem ao OE1 (Identificar os principais desafios enfrentados durante a migração de um sistema de software legado).

O primeiro desafio, está relacionado à identificação de funcionalidades. Não é uma tarefa trivial, especialmente em casos de grandes módulos através dos quais as funcionalidades são espalhadas e entrelaçadas entre si. Este é, na verdade, um problema recorrente, já discutido na literatura (OSSHER; TARR,2002).

O segundo desafio é a definição de limites ótimos entre os recursos candidatos para microserviços. Uma vez estabelecidos esses limites, segue-se o terceiro desafio,

[4] https://www.doctrine-project.org/projects/migrations.html

que é decidir quais serão convertidos em microserviços. Após essa decisão é enfrentado o quarto desafio, relacionado à análise criteriosa desses candidatos a microserviços quanto à sua respectiva granularidade e coesão.

A literatura já abordava o problema de decomposição para identificar módulos, pacotes, componentes e serviços "tradicionais", principalmente por meio de técnicas de *clustering* sobre artefatos de design ou código-fonte. No entanto, os limites entre os módulos definidos usando essas abordagens, eram muito flexíveis e permitiam que o software evoluísse para instâncias de *Big Ball of Mud* (COPLIEN; SCHMIDT, 1995).

## 4.5 LIÇÕES APRENDIDAS DO ESTUDO DE CASO

Embora tenha sido muito escrito sobre o valor dos serviços coesivos e o poder de contextos delimitados, parece haver um vazio na orientação sobre como identificá-los na prática (MCLARTY, 2017).

A questão principal é que as pessoas que tentam determinar os limites dos serviços são tecnólogos que procuram uma solução tecnológica, mas a definição de limites de serviço coesos alinhados à capacidade requer uma experiência do domínio. Para conseguir essa experiência deve ser realizado, independentemente da tecnologia usada, um exercício de modelagem. Portanto, há uma necessidade de orientação sobre como identificar o valor dos serviços coesos e o poder dos contextos delimitados (MCLARTY, 2017).

Alguns conceitos de DDD, mais notavelmente a idéia de contextos delimitados, ganharam popularidade e iniciaram a conversa do setor na definição da delimitação do serviço. É importante ressaltar que as abstrações de nível superior na metodologia DDD, como: domínios, subdomínios, contextos delimitados, agregados e mapas de contexto são agnósticos em relação à tecnologia e centrados no modelo (MCLARTY, 2017).

O mapa de contexto é um padrão estratégico do DDD, que pode ser um ponto de partida interessante para representar visualmente um sistema de software, já que uma possível abordagem para compreendê-lo é focar na relação entre seus

componentes. No entanto, no nível de mapeamento de contexto, o sistema de software completo de uma grande organização definitivamente não é trivial. Para tornar o mapa de contexto mais útil é importante ter um *"contexto"* para o próprio mapa de contexto. Esse contexto pode ser a decomposição de um aplicativo monolítico específico ou as interações de serviço de uma iniciativa específica (MCLARTY, 2017). A aplicação desta estratégia complementa bem com a metodologia ágil *Scrum*. Corresponde ao princípio de exploração e experimentação apresentado por Eric Evans (EVANS, 2004).

A aplicação das estratégias acima produziu múltiplos microserviços autônomos, cada um com seu próprio banco de dados. Para comunicação entre os microserviços usamos mecanismos de comunicação HTTP, como API *Restful* e também comunicação assíncrona com uma implementação EventBus, executando *RabbitMQ*[5]. Como mostrado na figura 10, cada um dos microserviços agora trabalha com um banco de dados relacional independente, exceto o serviço Marketing, pois é um serviço auxiliar. Para este, foi optado usar um banco de dados em memória.

## 4.6 TRABALHOS RELACIONADOS

Uma pesquisa relatada em (TAIBI; LENARDUZZI; PAHL, 2017) e conduzida entre profissionais experientes que já migraram seus monólitos para microserviços, descreveu que as capacidades de manutenção e escalabilidade foram classificadas como as motivações mais importantes. O mesmo artigo apresenta três características dos *frameworks* de processos adotados pelos entrevistados praticantes. Os dois primeiros processos se concentraram em executar a migração, reimplementando o sistema a partir do zero (TAIBI; LENARDUZZI; PAHL, 2017). O terceiro processo focalizou-se na implementação de apenas novos recursos como microserviços. Esses processos têm características em comum: análise da estrutura do sistema, definição da arquitetura do sistema e priorização do desenvolvimento de recursos/serviços (TAIBI; LENARDUZZI; PAHL, 2017).

---

[5] https://www.rabbitmq.com

Essas características estão de acordo com o roteiro proposto nesta seção. No entanto, neste caso, diferentemente do que foi relatado em (TAIBI; LENARDUZZI; PAHL, 2017), foram descritos os cenários e as etapas em que a migração ocorreu.

## 4.7 CONCLUSÃO DO CAPÍTULO

Este capítulo descreveu a migração do sistema eShop separando em dois cenários distintos. O primeiro cenário demonstrou a migração do sistema legado para uma versão modularizada, aplicando alguns dos conceitos do DDD. No segundo cenário, demonstrou como foi realizada a migração do sistema modularizado para uma arquitetura de micro- serviços. E ao final, apresenta as lições aprendidas durante a implantação deste estudo de caso, relacionando com trabalhos existentes na literatura acerca da decomposição e identificação de artefatos.

# 5 GUIA PROPOSTO

Este capítulo apresenta um guia para apoiar a migração de sistema de software legados monolíticos para arquitetura baseada em microserviços, que atende ao objetivo geral (OG) dessa pesquisa. As etapas do guia estão organizados em três fases principais, conforme ilustrado na figura 11.

## 5.1 CONTEXTO

Sistemas monolíticos legados geralmente apresentam problemas de modularidade, resultando em níveis significativos de acoplamento e baixa coesão. Rescrever um sistema a partir do zero, na maioria das vezes, é uma tarefa quase impossível. É recomendável portanto, efetuar uma migração de maneira gradual, substituindo partes específicas do sistema legado para novos módulos. Esse tipo de abordagem já é discutido no padrão *Strangler* (TAIBI; LENARDUZZI; PAHL,2017). À medida em que as funcionalidades  são migradas para novos módulos, ou um novo sistema, o sistema legado será totalmente "estrangulado", até o ponto em que não terá mais utilidade.

Após a conclusão deste processo, o sistema legado será abandonado. Portanto, é importante observar que essa abordagem ajuda a minimizar o risco durante a migração e distribui o esforço de desenvolvimento ao longo do tempo.

## 5.2 PREMISSAS PARA USO DO GUIA

A primeira condição para uso do guia proposto é a de que o sistema alvo tenha uma arquitetura monolítica, que seja possível identificar pelo menos três camadas, como: camada de apresentação, camada de negócio e camada de dados. É também necessário que o desenvolvedor ou a equipe de desenvolvimento tenha um mínimo de conhecimento das regras de negócios do sistema. Esse conhecimento será fundamental durante a execução das fases 1 e 2. Estas fases tem o foco em extrair o conhecimento do domínio e estabelecer limites coerentes com as regras de negócios do sistema.

Como o objetivo do guia é migrar um sistema com arquitetura legada monolítica para

uma arquitetura de microserviços, é necessário na parte operacional um nível de disciplina e algumas competências como descrito a seguir.

**Implantação Rápida**. Como a arquitetura de microserviços promove a criação de serviços independentes; é necessário automatizar a implantação destes serviços para tornar esta uma tarefa mais rápida, tanto para equipes quantos ambientes diferentes. Por exemplo: ambientes de desenvolvimento, testes e produção.

**Provisionamento de Ambientes**. Dada a necessidade de automatizar o processo de implantação, surge naturalmente a necessidade de um rápido provisionamento ambientes, o que se encaixa muito bem com *Cloud Computing*.

## 5.3 ESTRUTURA DESTE GUIA

A fase 1 deste guia, foca na análise e identificação das principais funcionalidades e suas respectivas responsabilidades. A fase 2 tem o foco em detalhar as características preexistentes do sistema monolítico, usando alguns dos conceitos chave do Domain-Drive Design (DDD), como: Contexto Delimitado, Mapa de Contexto, Domínios, Subdomínios, Agregador, Objetos de Valor e Serviços de Domínio. Estes conceitos ajudam a construir artefatos que apoiam as decisões relativas à migração a ser executada, incluindo a granula- ridade e coesão dos serviços a serem implementados. A última fase deste guia, portanto a fase 3, promove a migração dos contextos delimitados que foram identificados e mapeados na fase anterior para uma arquitetura de microserviços.

## 5.4 FASE 1: SOFTWARE MONOLÍTICO

**Etapa 1:** *Analisar o Código-Fonte e o Banco de Dados*

**Dados de Entrada:** *Código-Fonte e o Banco de Dados*

**Dados de Saída:** *Código-Fonte e o Banco de Dados*

**Descrição da Etapa 1:** A primeira fase deste guia é iniciado pelo processo de identificação dos artefatos no sistema. Esta etapa analisa o código-fonte e o banco de dados do sistema legado. Para realizar essa análise é necessário navegar entre

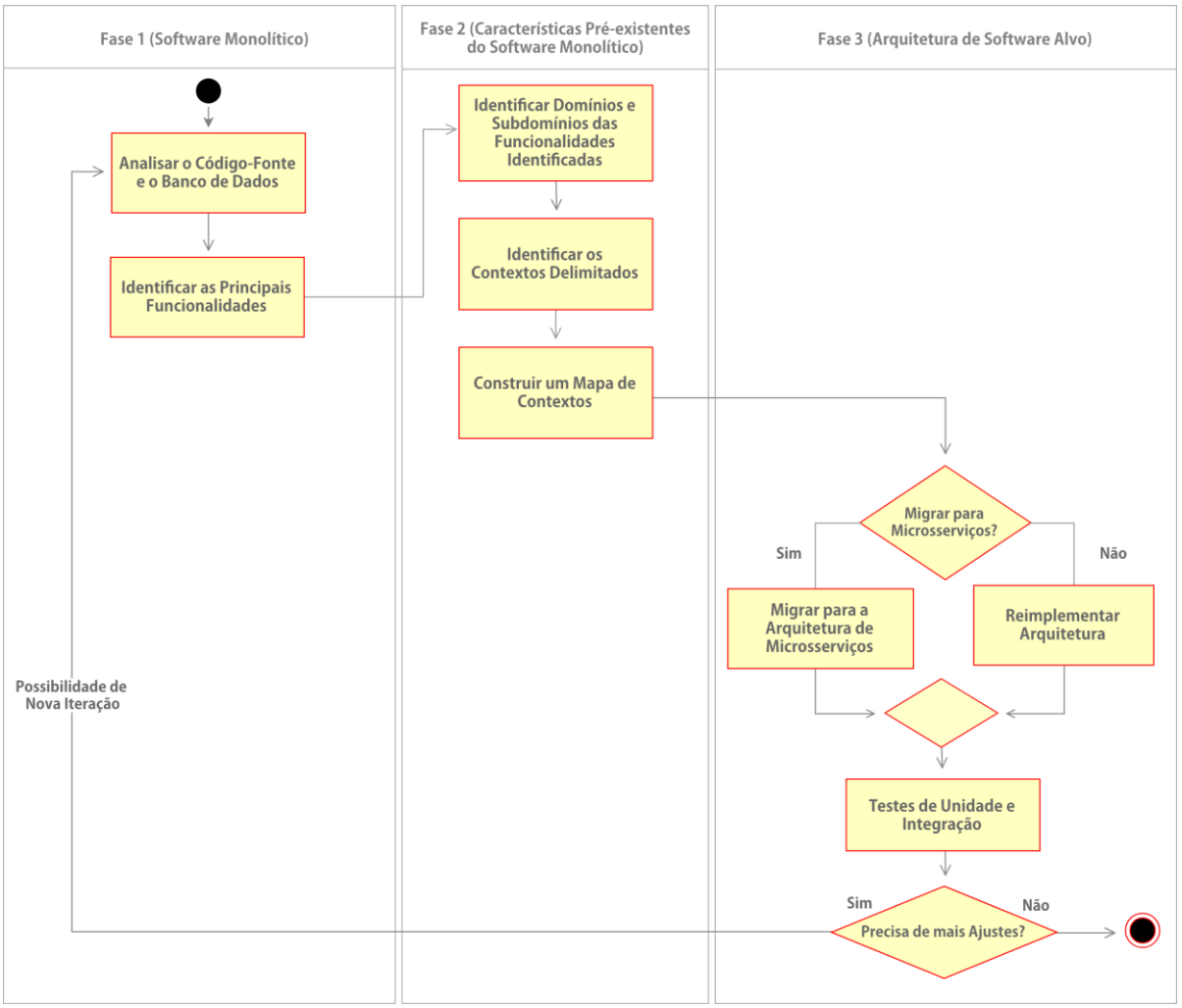
os arquivos e diretórios do sistema, para obter mais clareza do domínio a ser trabalhado.

**Etapa 2:** *Identificar as Principais Funcionalidades*

**Dados de Entrada:** *Código-Fonte e o Banco de Dados*

**Dados de Saída:** *Lista de Funcionalidades Identificadas*

Figura 11 – Guia Proposto




**Descrição da Etapa 2:** O objetivo desta etapa é identificar os propósitos, responsabilidades e as principais funcionalidades dos artefatos. É importante ressaltar que este é um processo iterativo e incremental e neste primeiro momento é essencial documentar, por mais que de forma simples. Com uma lista simples, apenas para registrar o que foi identificado durante a navegação nos artefatos do

sistema. O próximo passo é estabelecer um limite provisório entre estas funcionalidades, onde o objetivo é garantir mais clareza e compreensão das regras de negócio do sistema.

## 5.5 FASE 2: CARACTERÍSTICAS PRÉ-EXISTENTES DO SOFTWARE MONOLÍTICO

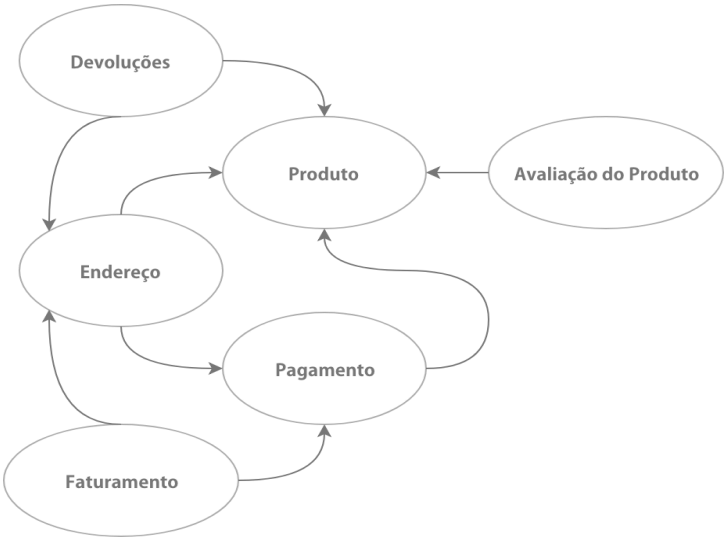
**Etapa 3:** *Identificar Domínios e Subdomínios das Funcionalidades Mapeadas*

**Dados de Entrada:** *Lista de Funcionalidades Identificadas*

**Dados de Saída:** *Diagrama com os Subdomínios Identificado***Descrição da Etapa 3:** Nesta etapa, o objetivo é destilar o domínio para proporcionar um conhecimento cada vez mais profundo. Por isso, é importante criar um modelo de domínio. O modelo de domínio é um modelo abstrato de alto nível, que revela e organiza os dados de conhecimento do domínio com a intenção de fornecer uma linguagem clara para desenvolvedores e especialistas de domínio. Esse esforço de identificação das funcionalidades, pode ser colaborativo envolvendo o time de desenvolvimento e especialistas de domínio e *stakeholder's* quando houver. É importante esboçar um simples diagrama, sem formalidades, pois o objetivo é ter clareza e aumentar o conhecimento das funções de negócios do domínio.

]

Figura 12 – Esboço de um Diagrama Simples com os Subdomínios Identificados




É muito provável que o código-fonte esteja acoplado à estrutura da aplicação Web, por isto a tarefa de identificação das funcionalidades precisa ser revisada a todo tempo, para que os limites de domínios e subdomínios sejam validados e compreensivos.

No diagrama do modelo de domínio ilustrado na Figura12, inclui objetos do mundo real como: Devoluções, Produto, Endereço, Pagamento, entre outros. Esses objetos podem possuir comportamentos distintos em determinados momentos, portanto algumas características do Produto e do Pagamento podem variar, ou seja, no momento do pagamento de um produto a única informação importante para realizar a operação é a identificação do produto. Por isso, pode ser mais interessante criar modelos distintos que representam o mesmo objeto do mundo real. Desta forma, cada modelo pode atender as necessidades específicas do seu contexto.

**Etapa 4:** *Identificar os Contextos Delimitados*

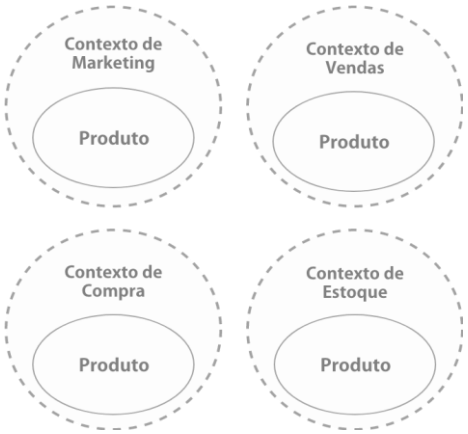
**Dados de Entrada:** *Diagrama com os Domínios e Subdomínios*

**Dados de Saída:** *Contextos Delimitados  Identificados*

**Descrição da Etapa 4:** Como ilustrado na Figura13com o conceito de Con- texto Delimitado é possível estabelecer limites a um domínio específico de acordo com as

intenções de negócios (Marketing, Vendas, Compra e Estoque).

Figura 13 – Exemplo de um Domínio Dividido em Quatro Contexto Delimitados




A medida que os domínios e subdomínios forem sendo identificados e preenchidos no diagrama é importante classificar as funcionalidades essências para o negócio e o relacionamento existente entre as demais. É importante deixar claro que ainda nesta etapa não deve haver qualquer preocupação com os detalhes de implementação. O foco ainda é no conhecimento do domínio.

**Etapa 5:** *Construir um Mapa de Contexto*

**Dados de Entrada:** *Contextos Delimitados Identificados*

**Dados de Saída:** *Mapa de Contextos*

**Descrição da Etapa 5:** Feito o mapeamento dos Contexto Delimitados, o pró- ximo passo é construir um Mapa de Contextos. O intuito deste mapa é tornar explícito o entendimento dos contextos e os relacionamentos entre eles. Assim como o mapeamento dos Contextos Delimitados, o Mapa de Contexto também precisa ter um processo con- tínuo de melhoria, para que cada vez mais as informações dos contextos delimitados e consequentemente do mapa de contextos sejam aprimoradas.

Nesta etapa da migração é possível identificar com base na modelagem tática do DDD, possíveis candidatos a entidades, objetos de valor, serviços, entre outros.

O principal foco da primeira e segunda fase é a extração de conhecimento do

domínio, identificando as principais funcionalidades e responsabilidades em vista de um estabelecimento de limites coerentes e validados com base nas regras de negócio do sistema.

## 5.6 FASE 3: ARQUITETURA DE SOFTWARE ALVO

Nessa fase final, o foco está na decisão e implementação da arquitetura. A primeira etapa desta fase é decidir se a arquitetura do sistema será migrada para uma arquitetura de microserviços. Esta é uma decisão que cabe a equipe de desenvolvimento, que a partir do mapa de contexto definido, pode avaliar a complexidade do sistema, e se há realmente a necessidade de migrar para uma arquitetura baseada em microserviços.

Como descrito no capítulo do estudo piloto, foi percebido durante a análise do mapa de contextos que o sistema tinha apenas três contextos: *Identity Context*, *OfferContext* e *TicketContext*. Por se tratar de um sistema simples, com um número reduzido de contextos, optou-se então por manter a arquitetura monolítica, tendo em vista o avanço consquistado na modularização através da aplicação dos conceitos do DDD. Um dos motivos para decisão de não optar pela migração para uma arquitetura de microserviços foi o aumento da complexidade necessária para implantanção dessa arquitetura. Portanto, o número de contextos é um critério que pode ser utilizado para a tomada de decisão. Entretanto, não é possível estabelecer um valor do número de contexto.

**Etapa 6:** *Migrar para a Arquitetura de Microserviços*

**Dados de Entrada:** *Sistema com Arquitetura Monolítica Evoluída*

**Dados de Saída:** *Sistema com Arquitetura de Microserviços*

**Descrição da Etapa 6:** Caso a decisão tenha sido em favor de migrar para a arquitetura de microserviços, em seguida iniciará uma série de ações. Mas antes é importante observar que em um cenário de uma grande aplicação onde o domínio é complexo é muito comum ter muitos contextos. Decidir por qual contexto iniciar a implementação não é uma tarefa trivial. Uma estratégia eficiente para lidar com este desafio, pode ser feita utilizando a seleção de contextos que possuem nenhum ou o

menor número de relacionamentos. O número de *bugs* associados a um determinado contexto também pode ser um outro critério a ser considerado para classificar os contextos candidatos a iniciar a migração.

Portanto, para cada contexto delimitado deve ser criado um diretório, dentro de cada um dos diretórios, três novos diretórios devem ser adicionados, sendo um para cada camada: Domain, Application, Infrastructure. Eles devem conter o código-fonte necessário para que o contexto delimitado funcione.

É crucial considerar os modelos de domínio e seus invariantes e reconhecer *Entities*, *Value Object's* e também *Aggregates*. O código-fonte deve ser mantido nesses diretórios conforme descrito na sequência. O diretório Application deve conter todos os serviços de aplicação e manipuladores de comandos.

O diretório Domain contém as classes com padrões táticos existentes no DDD, como: Entity, Value Object, Domain Event, Repository e Factory. O diretório Infrastructure deve fornecer os recursos técnicos para outras partes do aplicativo, isolando toda a lógica de domínio dos detalhes da camada de infraestrutura.

A camada de infraestrutura deve conter em detalhes, o código para enviar emails, enviar mensagens, armazenar informações no banco de dados, processar solicitações HTTP e fazer solicitações para outros servidores. Qualquer estrutura e biblioteca relacionada ao mundo externo, como sistemas de rede e de arquivos, deve ser usada ou chamada na camada de infraestrutura.

A estrutura de diretórios de um contexto delimitado deve ser organizado da seguinte maneira:

```
+--src
|          +-- SistemaLegado
|             +-- Contexto
|                +-- Application
|                +-- Domain
|                +-- Infrastructure
+-- tests
```

No cenário onde deve acontecer a migração da arquitetura para microserviços, os

contextos delimitados terão papel fundamental para organizar e identificar os microserviços (NADAREISHVILI et al., 2016).

É imprescindível portanto, que cada serviço possua sua própria estrutura permitindo assim a manutenção separada de repositórios externos. Isso facilita a evolução e os ajustes da implementação, que podem ser realizados separadamente, evitando possíveis efeitos colaterais (SPoF) em outros serviços. É prudente organizar os contextos em camadas bem definidas, pois desta forma permite expressar o modelo de domínio e a lógica de negócios, eliminando dependências na infraestrutura, interface do usuário e lógica da aplicação, conceitos que muitas vezes se misturam. Tudo que for relacionado ao modelo de domínio deve ser concentrado em uma camada, isolando-o das camadas superiores, como a camada de interface do usuário, camada de aplicação e de infraestrutura (EVANS, 2004).

**Etapa 7:** *Executar Testes de Unidade e Integração*

**Dados de Entrada:** *Contexto Delimitado Migrado*

**Dados de Saída:** *Contexto Delimitado Migrado com Testes Executados*

**Descrição da Etapa 7:** À medida que os contextos delimitados forem sendo migrados, é recomendado que os testes automatizados sejam executados, inicialmente testes unitários. A intenção é garantir que as partes implementadas estejam funcionando conforme esperado.

Essas três fases formam o conjunto mínimo principal para realizar uma migração. Existe a possibilidade de incluir novos passos em uma ou mais das três fases, dependendo das características específicas do sistema de software em questão.

## 5.7 LIMITAÇÕES DO GUIA

Uma das limitações deste guia foi ter sido aplicado somente na linguagem PHP.

## 5.8 CONCLUSÃO DO CAPÍTULO

Este capítulo descreveu, de maneira sequencial, as etapas do guia proposto para

realizar a migração de um sistema de software monolítico legado para arquitetura de microserviços.

# 6 CONCLUSÃO E PERSPECTIVAS FUTURAS

Este capítulo apresenta as considerações finais, contribuições, e as perspectivas de trabalhos futuros.

## 6.1 CONSIDERAÇÕES FINAIS

A migração de um sistema legado é geralmente um trabalho difícil e complexo, raramente pode ser realizado sem esforço significativo. Até onde sabemos, existem estruturas que podem ser usadas para apoiar os profissionais durante o desenvolvimento (*forward engineering*) de sistemas baseados em microsserviços, como Spring Cloud[6] [1], Hystrix[7] [2] e Istio[8][3], só para citar alguns. No entanto, nenhum deles fornece suporte completo para as três fases de migração representadas na Figura11. Para preencher essa lacuna, esta dissertação propôs um guia para apoiar a migração de sistemas de software legados para microserviços.

Acredita-se que a disponibilidade desse guia pode apoiar e incentivar profissionais da indústria e da academia a realizar esse tipo de migração. É importante ressaltar que a efetividade deste guia, necessita ser melhor avaliada, considerando que nesta dissertação a avaliação ocorreu com base nos resultados do estudo de caso apresentado no capítulo 4.

O capítulo 3 descreveu o sistema ePromo e as configurações do estudo exploratório composto em duas fases. Explorou as etapas da migração do sistema legado para uma arquitetura melhor modularizada, com o apoio das fases sugeridas no guia proposto. Ao final, apresentou as lições aprendidas, com base na experiência adquirida durante a execução deste estudo piloto.

O capítulo 4 descreveu o processo de migração do sistema eShop, separando em dois cenários distintos. No primeiro cenário foi demonstrado a migração do sistema legado para a versão modularizada, aplicando os conceitos do DDD. Em seguida, detalhou o segundo cenário, onde foi realizado a migração da versão modularizada para uma arquitetura de microserviços. Por fim, apresentou as lições aprendidas

[6] http://projects.spring.io/spring-cloud
[7] https://github.com/Netflix/Hystrix
[8] https://istio.io

durante a implantação do estudo de caso e a relação com trabalhos existentes na literatura acerca de decomposição e identificações de artefatos.

Por fim, no capítulo 5, foi apresentado em detalhes, de maneira sequencial as etapas do guia proposto. As etapas foram organizadas em três fases principais, como ilustrado na figura 11.

## 6.2 CONTRIBUIÇÕES

A principal contribuição desta dissertação foi compartilhar um guia que propõem através da experiência adquirida com estudos exploratórios, auxiliar profissionais durante a etapa de migração de sistema de software monolítico legado para arquitetura baseada em microserviços. As seguintes contribuições:

a) Divulgação das lições aprendidas com a execução dos estudos exploratórios.

b) Mostrar que, em um cenário de sistema simples, migrar para uma arquitetura baseada em microserviços só irá acarretar no aumento de esforço e complexidade para implantação e gerenciamento do sistema.

c) Sugerir uma estratégia para apoiar a tomada de decisão de qual contexto deve ser iniciada a implementação.

## 6.3 TRABALHOS EM ANDAMENTO E FUTUROS

É planejado aplicar este guia proposto em sistemas de software monolítico legado implementados em outras linguagens, diferentes do PHP. Também é planejado realizar uma pesquisa com profissionais da indústria para caracterizar sua percepção em relação aos desafios enfrentados durante esse tipo de migração, características dos possíveis processos que eles podem ter usado para esse propósito e opiniões sobre o roteiro proposto.

## REFERÊNCIAS[9]

AMARAL, M. et al. Performance evaluation of microservices architectures using containers. In: IEEE. *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*. [S.l.], 2015. p. 27–34.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: migration to a cloud-native architecture. *IEEE Software*, IEEE, v. 33, n. 3, p. 42–52, 2016.

BOKHARI, S. M. A. et al. Limitations of service oriented architecture and its combination with cloud computing. *Bahria University Journal of Information & Communication Technology*, Bahria University, v. 8, n. 1, p. 7, 2015.

COPLIEN, J. O.; SCHMIDT, D. C. *Pattern languages of program design*. [S.l.]: ACM Press/Addison-Wesley Publishing Co., 1995.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In: *Present and Ulterior Software Engineering*. [S.l.]: Springer, 2017. p. 195–216.

ERL, T. *Soa: principles of service design*. [S.l.]: Prentice Hall Upper Saddle River, 2008.v. 1.

EVANS, E. *Domain-driven design: tackling complexity in the heart of software*. [S.l.]: Addison-Wesley Professional, 2004.

FOWLER, M.; LEWIS, J. Microservices a definition of this new architectural term. *URL: http://martinfowler.com/articles/microservices.html.[Online*, 2014.

GAMMA, E. *Design patterns: elements of reusable object-oriented software*. [S.l.]: Pearson Education India, 1995.

GARCIA, D. F. et al. Experimental evaluation of horizontal and vertical scalability of cluster-based application servers for transactional workloads. In: *8th International Conference on Applied Informatics and Communications (AIC'08)*. [S.l.: s.n.], 2008. p. 29–34.

GROVES, D. Successfully planning for soa. *BEA Systems Worldwide*, v. 11, 2005.

HUHNS, M. N.; SINGH, M. P. Service-oriented computing: Key concepts and principles. *IEEE Internet computing*, IEEE, v. 9, n. 1, p. 75–81, 2005.

HUTCHINSON, J. et al. Evolving existing systems to service-oriented architectures: Perspective and challenges. In: IEEE. *Web Services, 2007. ICWS 2007. IEEE International Conference on*. [S.l.], 2007. p. 896–903.

JADEJA, Y.; MODI, K. Cloud computing-concepts, architecture and challenges. In: IEEE. *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*. [S.l.], 2012. p. 877–880.

[9] De acordo com a Associação Brasileira de Normas Técnicas NBR 6023.

KALSKE, M.; MäKITALO, N.; MIKKONEN, T. Challenges when moving from monolith to microservice architecture. In: *Current Trends in Web Engineering*. Springer, Cham, 2017. p. 32–47. Disponível em: <https://link.springer.com/chapter/10.1007/ 978-3-319-74433-9_3>.

KWAN, A. et al. Microservices in the modern software world. In: *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*. Riverton, NJ, USA: IBM Corp., 2016. (CASCON '16), p. 297–299. Disponível em:<http://dl.acm.org/citation.cfm?id=3049877.3049915>.

LEYMANN, F. et al. Native cloud applications: Why monolithic virtualization is not their foundation. In: *Cloud Computing and Services Science*. Springer, Cham, 2016. p. 16–40. Disponível em:<https://link.springer.com/chapter/10.1007/978-3-319-62594-2_2>.

MAHMOOD, Z. The promise and limitations of service oriented architecture. *International journal of Computers*, Citeseer, v. 1, n. 3, p. 74–78, 2007. MARTIN, R. C. The single responsibility principle. *The principles, patterns, and practices of Agile Software Development*, v. 149, p. 154, 2002.

MARTIN, R. C. *The Single Responsibility Principle*. 2014.

MARTIN, R. C. The clean architecture. 2012. *URL https://8thlight. com/blog/uncle-bob/2012/08/13/the-clean-architecture. html*, 2017.

MCLARTY, M. *Designing a Microservice System*. 2017.

MILLETT, S. *Patterns, Principles and Practices of Domain-Driven Design*. [S.l.]: John Wiley & Sons, 2015.

NADAREISHVILI, I. et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. [S.l.]: "O'Reilly Media, Inc.", 2016.

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

OSSHER, H.; TARR, P. Multi-dimensional separation of concerns and the hyperspace approach. In: *Software Architectures and Component Technology*. [S.l.]: Springer, 2002. p. 293–323.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. In: *CLOSER (1)*. [S.l.: s.n.], 2016. p. 137–146.

PAPAZOGLOU, M. P. et al. Service-oriented computing: State of the art and research challenges. *Computer*, IEEE, v. 40, n. 11, 2007.

RICHARDSON, C. *Monolithic Architecture*. 2015.

SANTIS, L. F. S. D.; ROSA, E. *Evolve the Monolith to Microservices with Java and Node*. [S.l.]: IBM Redbooks, 2016.

SEAMAN, C. B. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, IEEE, v. 25, n. 4, p. 557–572, 1999.

SINGLETON, A. The economics of microservices. *IEEE Cloud Computing*, IEEE, v. 3, n. 5, p. 16–20, 2016.

SRINIVASAN, L.; TREADWELL, J. An overview of service-oriented architecture, web services and grid computing. *HP Software Global Business Unit*, Citeseer, v. 2, 2005.

TAIBI, D.; LENARDUZZI, V.; PAHL, C. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, IEEE, v. 4, n. 5, p. 22–32, 2017.

TOFFETTI, G. et al. An architecture for self-managing microservices. In: ACM. *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. [S.l.], 2015. p. 19–24.

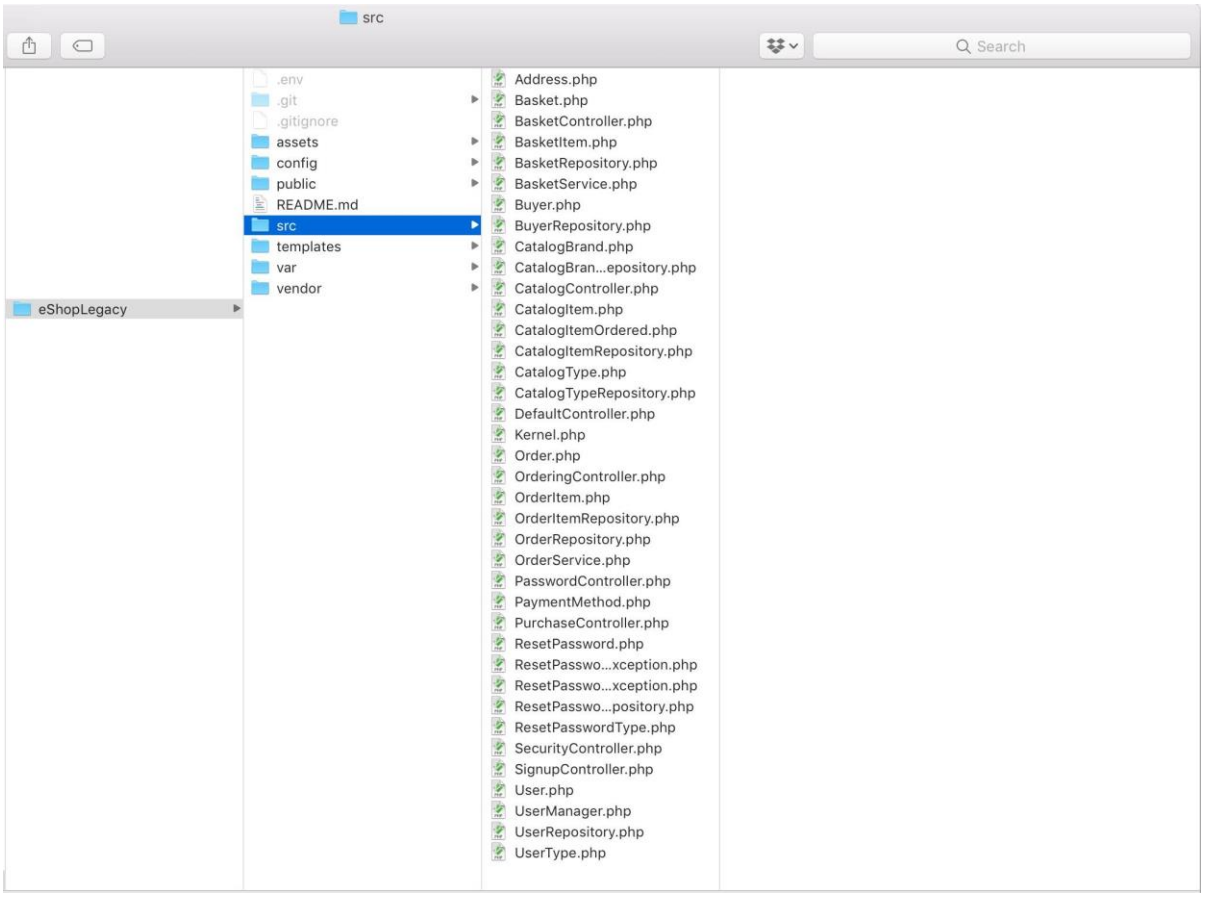
TOFFETTI, G. et al. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, Elsevier, v. 72, p. 165–179, 2017.

WOLFF, E. *Microservices: Flexible Software Architecture*. [S.l.]: Addison-Wesley Professional, 2016.

ZDUN, U.; NAVARRO, E.; LEYMANN, F. Ensuring and assessing architecture conformance to microservice decomposition patterns. In: SPRINGER. *International Conference on Service-Oriented Computing*. [S.l.], 2017. p. 411–429.

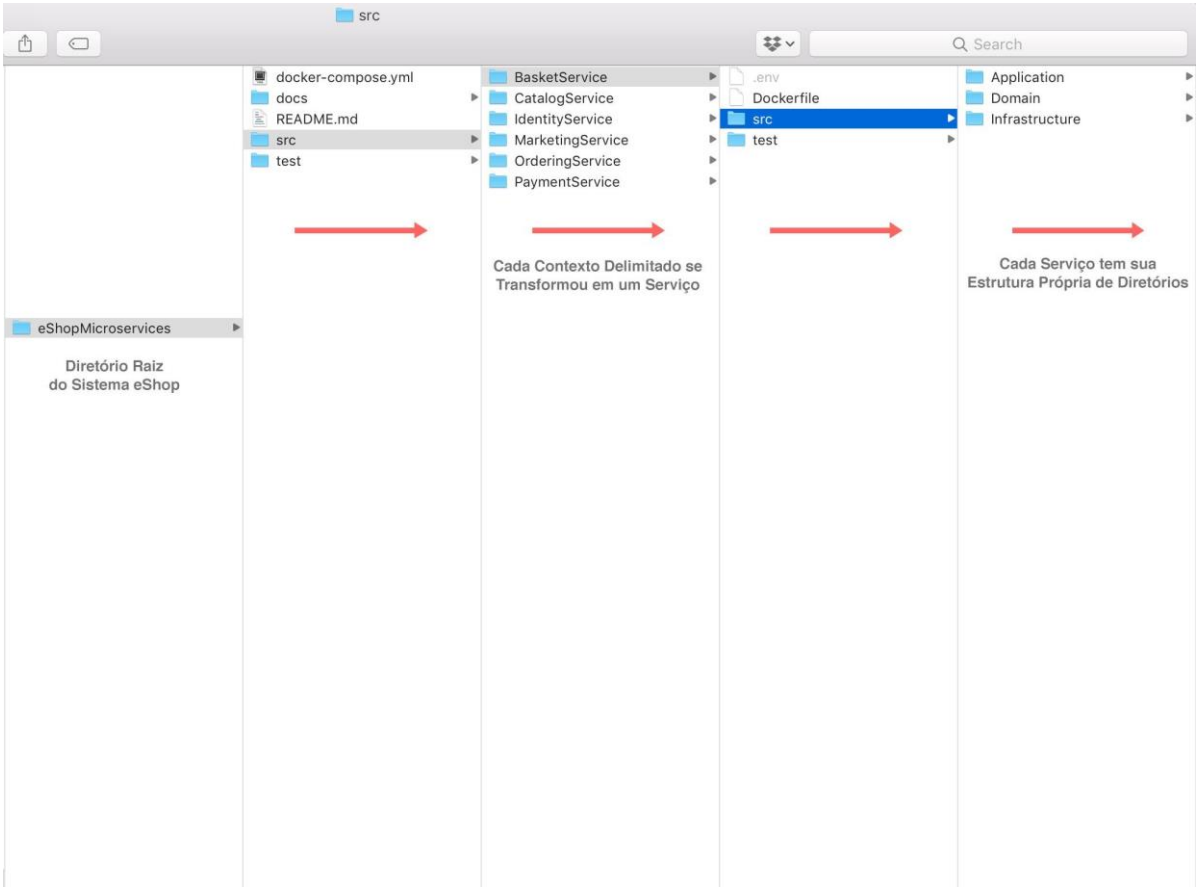
# ANEXO A – SISTEMA ESHOP LEGADO

Figura 14 – Uma Visão Geral da Estrutura de Arquivos e Diretórios do Sistema eShop



## Anexo B – SISTEMA ESHOP COM ARQUITETURA DE MICROSERVIÇOS

Figura 15 – Uma Visão Geral do Sistema eShop Após a Execução do Guia Proposto



Listing 1 – Uma Visão Geral do TicketsController (Estudo Piloto)

```
class TicketsController extends Controller
{
        //...
        public function postAction(Request $request): JsonResponse
        {
            $content = json_decode($request->getContent(),true);

               $timer  =  $this->getRepository('AppBundle:Timer')->find($content['timer']);
               $offer  =  $this->getRepository('AppBundle:Offer')->find($content['offer']);

               $customer = new Customer(
                    $content['customerName'],
                    $content['customerEmail'],
                    $content['customerPhone']
            );

               $em = $this->getEntityManager();
               $em->persist($customer);

               if ($offer->getTicketCode() === null || $offer->getTicketCode() === 'auto') {
                     $ticketCode =substr(sha1(uniqid(mt_rand(),true)), 0, 8);
            }

            $ticket = new Ticket($offer, $customer, $timer, $offer->getTicketCode());

            $this->get('ticket.validator')->validate($ticket);

               $em->persist($ticket);
               $em->flush();

            $company  =  $this->getRepository('AppBundle:Company')->findOneBy(['user' =>  $offer-
>getOwner()]);

               $message = (new Message())
                     ->setFrom('noreply@example.com')
                     ->setTo($ticket->getCustomer()->getEmail())
                     ->setBody(
                           $this->view(
                                 ':mailer:ticket_created.html',
                              [
                                          'ticket' => $ticket,
                                          'offer' => $offer,
                                          'company' => $company
                              ]
                        ),
                        'text/html'
                  );

            $this->get('mailer')->send($message);

            return new JsonResponse(['ticketId' => $ticket->getId()], 201);
        }
}
```

Listing 2 – As regras de negócio que estavam no controlador agora são executadas

pelo manipulador do comando de criação do Ticket (Pilot Study).

```
final class CreateTicketHandler
{
        //...
        public function invoke(CreatingTicket $command): void
        {
                $timer = $this->timerRepository->fromId($command->timerId());
                $offer = $timer->getPoint()->getOffer();

    $ticketCode = $offer->getTicketCode();

                if ($ticketCode === null || $ticketCode === 'auto') {
                        $ticketCode = substr(sha1(uniqid(mt_rand(),true)), 0, 8);
                }

                $ticketId = $command->ticketId();

                $ticket = new Ticket(
                        $ticketId,
                        $command->offerId(),
                        $command->customer(),
                        $timer,
                        $ticketCode
    );

                $this->ticketValidator->validate($ticket);
                $this->ticketRepository->save($ticket);

                $this->eventBus->handle(new TicketWasCreated($ticketId));
        }
}
```