# Capítulo 1

# Construindo Aplicações Distribuídas com Microsserviços

Luís Henrique Neves Villaça, Antônio Francisco Pimenta Jr. e Leonardo Guerreiro Azevedo

***Abstract***

*Solutions in a microservice architecture are developed using software component compositions. This approach is based on a set of services developed and deployed in independent of each other. They can be developed using different programming languages and use the technologies that best fit their needs. One of the main challenge in that approach is to integrate data coming from distinct DBMSs (e.g., relational, spatial, NoSQL), which configures an architecture of polyglot database. This work presents how to create applications using microservices architecture. Besides, it presents solutions that handle the polyglot database problem. As a result, it helps to identify situations where to apply a microservices architecture.*

***Resumo***

*Soluções em microsserviços são construídas por meio de composição de componentes de software. Sua abordagem arquitetural baseia-se em um conjunto de serviços que podem ser desenvolvidos e implantados independentemente uns dos outros. Eles podem ser escritos em diferentes linguagens de programação e usar as tecnologias de banco de dados que melhor atendam às suas necessidades. Um dos principais desafios desta abordagem é integrar dados provenientes de SGBDs distintos (por exemplo, relacional, espacial, NoSQL), o que configura uma arquitetura de cando de dados poliglotas. Este trabalho apresenta como construir aplicações empregando arquitetura de microsserviços. Além disso, soluções para o problema de bancos de dados poliglotas são apresentadas. Como resultado, este trabalho ajuda a identificar situações onde aplicar a arquitetura de microsserviços.*

## 1.1. Introdução

A arquitetura de microsserviços é uma abordagem para o desenvolvimento de uma única aplicação formada por um conjunto de serviços aplicados em um domínio limitado (microsserviços), cada um rodando em seu próprio processo (isolamento - contêiner) e se comunicando através de um mecanismo como uma API HTTP. Esses serviços são construídos em torno de uma parte específica do negócio e são implantados de forma completamente automatizada (Automação e Docker) [Fowler and Lewis 2014]. Cada um desses serviços implementa funções específicas, como, por exemplo, recomendações de produtos e gerenciamento de carros de compra, e possui um alto grau de autonomia (seu próprio banco de dados) para permitir sua evolução independente dos demais.

Um desafio dessa abordagem surge a partir da necessidade de integrar dados de origens distintas. Isso é um problema usual em muitas empresas, que já possuem diversas fontes de informação, como bancos de dados relacionais, NoSQL, planilhas, sistemas que gerenciam configurações de artefatos de software, e repositórios de dados não estruturados (configurando Persistência Poliglota). Um único relatório pode correlacionar informações de serviços oriundos de diversas fontes de dados. Nesse contexto, é importante que arquitetos, projetistas e desenvolvedores compreendam os aspectos envolvidos, a fim de avaliarem sua aplicabilidade em um determinado cenário [Sadalage and Fowler 2012].

Este trabalho tem objetivo de capacitar os leitores em pequisa e construção de aplicações empregando arquitetura de microsserviços. Além disso, ele se aprofunda no tema apresentando soluções para integração de dados em SGBDs distintos em uma arquitetura de microsserviços.

Este trabalho está dividido da seguinte forma. A Seção 1.2 apresenta os principais conceitos relacionados a SOA e Microsserviços. A Seção 1.3 apresenta persistência poliglota. A Seção 1.4 apresenta microsserviços e persistência poliglota na prática. Finalmente, a Seção 1.5 apresenta as considerações finais.

## 1.2. Fundamentação Teórica

Esta seção apresenta a fundamentação teórica sobre SOA e microsserviços.

### 1.2.1. Arquitetura Orientada a Serviços

Organizações modernas precisam responder de forma efetiva e rápida às oportunidades do mercado. Uma organização de médio e grande porte possui diversos departamentos (ou áreas), os quais em geral utilizam diferentes aplicações para realizar suas atividades. Estas aplicações necessitam se comunicar de forma integrada com o objetivo de atingir agilidade e simplificar processos de negócio, tornando-os mais produtivos, frente à crescente e a intensa competitividade do mercado.

O uso de serviços e de seus padrões de integração automatizada de negócios levou a grandes avanços na integração de aplicações. SOA (Service-Oriented Architecture ou Arquitetura Orientada a Serviços) é um paradigma para a realização e manutenção de processos de negócio em um grande ambiente de sistemas distribuídos que são controlados por diferentes proprietários [Josuttis 2007]. SOA é uma arquitetura conceitual onde funcionalidade do negócio, ou lógica da aplicação, é disponibilizada para usuários

SOA, ou consumidores, como serviços compartilhados e reutilizáveis em uma rede de TI [Marks and Bell 2008].

A nível conceitual, serviços são componentes de software providos através de um *endpoint* (ponto de acesso) acessível na rede [Vinoski 2002]. Serviços são módulos de negócio ou funcionalidades das aplicações que possuem interfaces expostas, e que são invocados via mensagens [Erl 2005]. Por exemplo, em um sistema bancário teríamos os seguintes serviços: serviço de nomes e endereços; serviço de abertura de conta; serviço de balanço de contas; serviço de depósitos. Serviços correspondem a recursos de software bem definidos através de uma linguagem padrão, são auto-contidos, proveem funcionalidades padrões do negócio, independentes do estado ou contexto de outros serviços [Erl 2005]. Serviço corresponde a uma representação lógica de uma atividade do negócio que pode ser mapeada em entrada, processamento e saída. Algumas características (ou princípios) de serviços são [Erl 2005]:

- Deve estar alinhado ao negócio, atendendo a uma necessidade representada em um processo da organização;
- Deve ser operacionalmente independente, garantindo alta coesão e acoplamento fraco;
- Deve fornecer os mesmos resultados para uma mesma entrada (isto é, ser sem estado ou *stateless*);
- Deve permitir composição;
- Deve ser atômico/auto-contido;
- Deve garantir consistência das informações;
- Devem ter pré e pós-condições bem definidas;
- Devem garantir interoperabilidade, ou seja, poder se comunicar com consumidores ou consumir outros serviços desenvolvidos empregando diversificadas tecnologias;
- Devem permitir reuso elevado.

Estes princípios podem ser relaxados de uma forma ou de outra para alcançar determinados objetivos como, por exemplo, serviços compostos os quais não são atômicos, pois utilizam outros serviços para realizar suas tarefas

Um serviço é implementado de acordo com um contrato, é exposto através de um *endpoint*, é governado por políticas, recebe mensagens de requisição e envia mensagens de resposta. Por outro lado, um cliente entende o contrato do serviço e estando aderente às políticas do serviço, liga-se a ele através do *endpoint* do serviço, e envia mensagens de requisição e recebe mensagens de resposta. Estes conceitos são ilustrados na Figura 1.1. A principal tecnologia de implementação serviços é a de web services, apresentada na Seção 1.2.2.

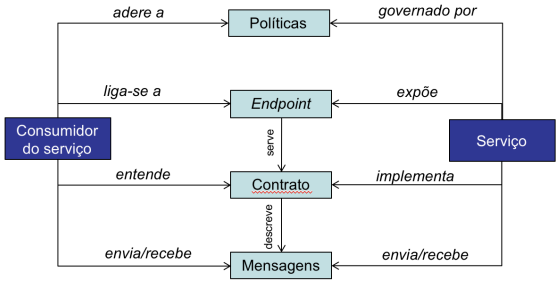



**Figura 1.1. Um exemplo de troca de mensagens entre consumidor e provedor**

Existe um terceiro elemento (ou parte interessada - *stakeholer*) na arquitetura orientada a serviços - o *service broker* ou registro de serviços. Ele atua como intermediário entre o provedor e o consumidor de serviço. O seu principal papel é prover informações de localização de serviço as quais estão contidas em um registro de serviços. O registro de serviços age como um diretório para serviços publicados, funcionando como um lista de páginas amarelas para números de telefones. Os provedores de serviço utilizam o registro para publicar informações sobre seus serviços e os consumidores de serviço usam o registro para procurarem por serviços. Apesar de *brokers* estarem associados com registros, eles pode realizar tarefas mais avançadas como, por exemplo, provisão de serviços de negociação e distribuição [Gu and Lago 2007].

A Figura 1.2 apresenta os três *stakeholders* de uma arquitetura orientada a serviços.

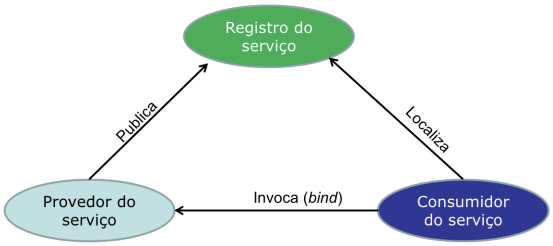



**Figura 1.2. SOA *stakeholders***

### 1.2.2. Web services

Um web service é um sistema de software projetado para apoiar interação interoperável máquina-a-máquina através de uma rede. Ele tem uma interface descrita em um formato processável por máquina. Outros sistemas interagem com o web service de uma maneira prescrita pelo formato de suas mensagens empregando um protocolo de comunicação (tipicamente HTTP) [Booths et al. 2004]. Existem dois tipos de web services: web service SOAP (também conhecidos como WS-* ou “Big” web services [Pautasso et al. 2008]) e web service RESTful.

#### 1.2.2.1. Web Services SOAP

Web Services SOAP empregam um conjunto de tecnologias baseadas em XML[1] (Extensible Markup Language), tais como SOAP, WSDL, XSD e UDDI.

XML[2] descreve uma classe de objetos de dados chamados de documentos XML e parcialmente descreve o comportamento de programas que os processam. Documentos XML são feitos de unidades de armazenamento chamadas de entidades, as quais contem caracteres "parsed" e caracteres não "parsed", alguns dos quais formam dados de caracteres e outros formam *markups*. *Markups* codificam uma descrição do *layout* e da estrutura lógica do armazenamento de documentos, provendo um mecanismo para definir restrições sobre os mesmos.

XSD (XML Schema Definition Language) oferece mecanismos para descrever a estrutura e restrições do conteúdo de documentos XML, incluindo aquele que exploram o mecanismos de Namespace. Namespace é um mecanismo para quebrar esquemas em subconjuntos de forma a obter definições reutilizáveis por mais de um projeto.

SOAP[3] (Simple Object Access Protocol) é um protocolo leve que tem o objetivo de troca de informações estruturadas em um ambiente descentralizado e distribuído. Ele usa as tecnologias XML para definir um *framework* de mensagens extensíveis, provendo um construto de mensagem que pode ser trocado sobre protocolos variados. Este *framework* foi projetado para ser independente de qualquer modelo de programação particular e outas semânticas específicas de implementação.

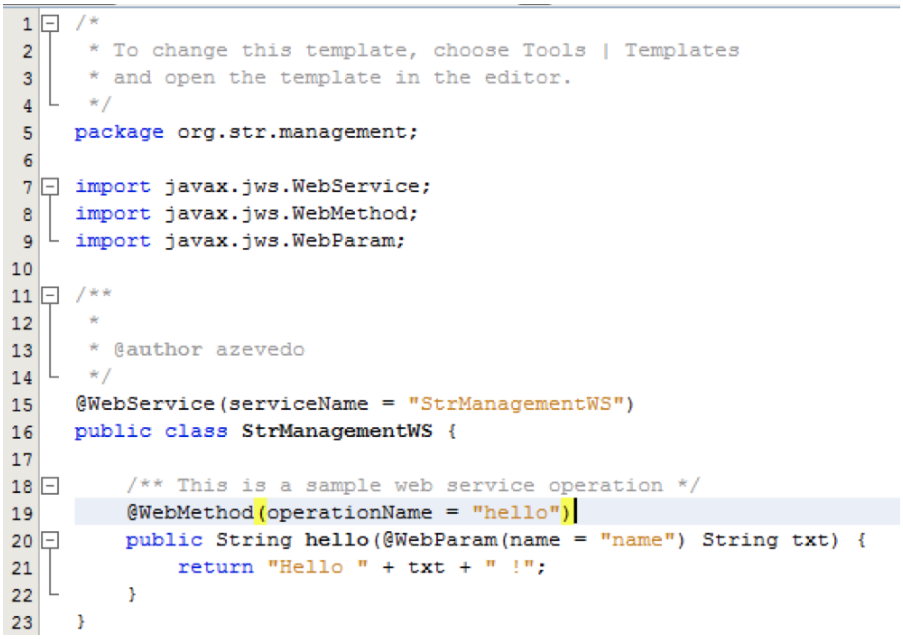
WSDL[4] (Web Service Description Language) define uma gramática XML para descrever serviços de rede como uma coleção de endpoints de comunicação capaz de realizar troca de mensagens. As definições de serviços em WSDL proveem documentação para sistemas distribuídos e serve como uma receita para automatizar os detalhes envolvidos na comunicação de aplicações.

A Figura 1.3 apresenta um exemplo de implementação de web service SOAP em Java. Nas linhas 7, 8 e 9 temos os *imports* necessários para as anotações @WebService, @WebMethod e @WebParam. O primeiro é utilizado (linha 15) para anotar a classe `StrManagementWS` como um web service SOAP disponibilizado como serviço `"StrManagementWS"`. O segundo é utilizado (linha 19) para anotar o método `hello` para ser invocado quando a operação `hello` for invocada. O terceiro é utilizado (linha 20) para definir o nome do parâmetro como sendo `name`.

#### 1.2.2.2. Web Service REST

Web service REST (Representational State Transfer) é uma alternativa mais simples a web services SOAP. Esta tecnologia é mais fácil de utilizar. É um estilo de projeto de aplicações mais coesas e menos acopladas que se baseia em recursos nomeados ao invés

[1] https://www.w3.org/TR/xml/
[2] https://www.w3.org/TR/xml/
[3] https://www.w3.org/TR/soap12-part1/
[4] https://www.w3.org/TR/wsdl/



```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package org.str.management;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

/**
 *
 * @author azevedo
 */
@WebService(serviceName = "StrManagementWS")
public class StrManagementWS {

    /** This is a sample web service operation */
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hello " + txt + " !";
    }
}
```

**Figura 1.3. Exemplo de implementação de web service SOAP em Java.**

de mensagens. Nesta tecnologia, serviços são vistos como recursos e podem ser identificados unicamente por suas URLs.

REST foi introduzido em 2000 por Roy Fielding em sua tese de doutorado na Universidade de Califórnica, Irvine [Fielding 2000]. Não atraiu muita atenção na época, mas hoje é amplamente utilizado como, por exemplo, pelo Yahoo, Google e Facebook.

REST[5] é apresentado pela W3C como sendo um subconjunto da Web baseado em HTTP nos quais agentes proveem semântica de interface uniforme - essencialmente criar, recuperar, atualizar e apagar - ao invés de interface arbitrárias ou específicas de aplicação, e manipulam recursos apenas pela troca de representações. Além disso, as interações REST são sem estado (*stateless*) no sentido de que o significado da mensagem não depende do estado da conversação.

REST não impõe restrições no formato da mensagem, como SOAP para web services SOAP, mas sim apenas no comportamento dos componentes envolvidos. Dessa forma, o desenvolvedor tem maior flexibilidade em optar pelo formato de mensagem que atenda melhor as suas necessidades. Os formatos mais comuns são JSON[6] (Java Script Object Notation), XML e texto puro, mas em teoria qualquer formato pode ser usado. A principal característica de web services REST é usar explicitamente os métodos HTTP (POST, GET, PUT, DELETE) para denotar a invocação de diferentes operações.

Web services REST se baseiam nos seguintes princípios: (i) Usar explicitamente os métodos HTTP; (ii) Ser sem estado; (iii) Expor URIs como uma estrutura de diretório; (iv) Transferir XML, JSON, ou ambos.

A Figura 1.4 apresenta um exemplo de implementação de web service REST em

[5]https://www.w3.org/TR/ws-arch/\#relwwwrest
[6]https://www.json.org/

Java. Nas linhas 3 e 4 temos os *imports* necessários para as anotações @Path e @GET. O primeiro é utilizado (linha 7) para anotar a classe `HelloResource` como um web service REST disponibilizado no caminho "`\hello`". O segundo é utilizado (linha 10) para anotar o método `getInformation` para ser invocado quando o método HTTP GET for executado neste caminho.

```
1  package com.ibm.cloudoe.samples;
2
3  import javax.ws.rs.GET;
4  import javax.ws.rs.Path;
5
6
7  @Path("/hello")
8  public class HelloResource {
9
10     @GET
11     public String getInformation() {
12
13         return "Hello World!";
14
15     }
16 }
```

**Figura 1.4. Exemplo de implementação de web service REST em Java.**

### 1.2.3. GraphQL

A linguagem GraphQL[7] foi elaborada pelo Facebook com o objetivo de viabilizar buscas de dados, com um grau maior de flexibilidade e eficiência que o disponibilizado em serviços REST e SOAP [Ghebremicael 2017]. A especificação GraphQL é de código aberto desde 2015. Ela permite que o usuário solicite informações a partir de um esquema denominado IDL (Interface Definition Language). Este esquema define um formato com os parâmetros de requisição de informações.

A Figura 1.5 apresenta um exemplo de esquema, consulta e resultado em GraphQL. O esquema inclui um tipo *Projeto* com os campos *nome*, *slogan* e uma lista de usuários *colaboradores*. A consulta busca o projeto cujo nome é *GraphQL* e solicita o *slogan* deste projeto como dado de retorno. Como resultado, obtem-se o slogan "Uma linguagem de consultas para APIs".

| Esquema | Consulta | Resultado |
|---|---|---|
| `type Projeto {`<br>`  nome: String`<br>`  slogan: String`<br>`  colaboradores:`<br>`[Usuario]`<br>`}` | `{`<br>`  projeto(`<br>`  nome: "GraphQL"`<br>`  ) {`<br>`  slogan`<br>`  }`<br>`}` | `{`<br>`  "projeto": {`<br>`  "slogan":`<br>`"Uma linguagem de`<br>`consultas para`<br>`APIs"`<br>`  }`<br>`}` |

**Figura 1.5. Exemplo de esquema, consulta e resultados em GraphQL.**

[7] `http://graphql.org`

Consultas GraphQL não apenas acessam propriedades de um recurso, mas seguem as referências entre eles. Por exemplo, enquanto uma API REST requer carregar dados de múltiplas URLs, uma API GraphQL obtêm todos os dados em apenas uma requisição, resultando em processamento mais eficiente.

GraphQL cria uma API uniforme sem estar atrelada a uma tecnologia de armazenamento específica. Dessa forma, sendo uma tecnologia diretamente aplicável ao acesso em bancos de dados poliglotas.

### 1.2.4. Arquitetura de Microsservicos

Esta seção apresenta os principais conceitos de microsserviços.

#### 1.2.4.1. Definição

A arquitetura de microsserviços (ou simplesmente microsserviços) surgiu empiricamente a partir de padrões arquiteturais utilizados no mundo real, onde sistemas são compostos por serviços que colaboram entre si para atingir seus objetivos, se comunicando a partir de mecanismos leves (como Web APIs) [Fowler and Lewis 2014, Newman 2015].

Fowler e Lewis definem **Arquitetura de Microserviços** como uma abordagem para o desenvolvimento de uma única aplicação formada por um conjunto de pequenos serviços (microsserviços), cada um rodando em seu próprio processo (isolamento - contêiner) e se comunicando através de algum mecanismo de pequeno porte, geralmente uma API HTTP. Esses serviços são construídos em torno de uma parte específica do negócio (DDD - Domain-Driven Design) e são implantados de forma completamente automatizada (Automação e Docker). Eles podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias de banco de dados. Existe uma camada mínima centralizada de gerenciamento desses serviços [Fowler and Lewis 2014].

#### 1.2.4.2. Arquitetura Monolítica x Arquitetura de Microsserviços

Microsserviços são uma alternativa às grandes aplicações monolíticas. A Figura 1.6 apresenta um exemplo de arquitetura monolítica [Richardson 2015]. A aplicação, na parte central, contém toda a lógica do negócio, a qual pode ser implementada através de módulos que definem serviços, objetos do negócio, e eventos. Esta aplicação se conecta a um SGBD (Sistema de Gerenciamento de Banco de Dados) para consultas e armazenamento de dados e a outros serviços externos como, por exemplo, serviço de consulta de CEP.

Apesar de ter uma arquitetura interna dividida em módulos, a aplicação é empacotada e disponibilizada como uma aplicação monolítica, por exemplo, um WAR[8] para arquivos Java ou aplicações Java empacotadas como executáveis JAR.

Estes tipos de aplicações são fáceis de desenvolver por IDEs e outras ferramentas focadas em construir uma única aplicação. Elas também são fáceis de serem testadas

[8]Web application ARchive (WAR) é um formato de arquivo para distribuir, por exemplo, uma coleção de JavaServer Pages, Servlets Java, classe Java para ser implantado em servidor de aplicação como o Tomcat ou Glassfish.

e distribuídas. Inclusive elas podem ser escalonadas colocando-se múltiplas cópias em múltiplos empregando balanceamento de carga.

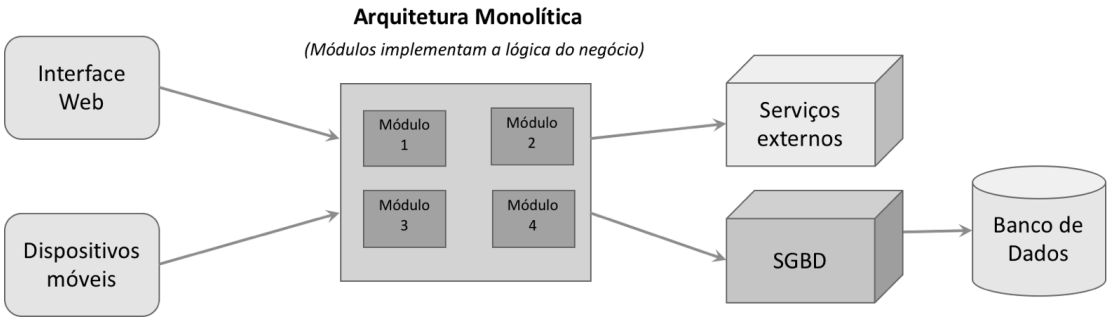



**Figura 1.6. Um exemplo de arquitetura monolítica (adaptado de [Richardson 2015])**

Problemas começam a surgir quando a aplicação "monolítica" cresce tornando-se extremamente complexa. Como resultado, corrigir erros (*bugs*) e implementar novas funcionalidades corretamente torna-se difícil e consome muito tempo. O problema se agrava se o código é difícil de entender, o que torna ainda mais complicado evoluir corretamente a aplicação . Além disso, aplicações muito grandes podem demorar a iniciarem. Se for necessário implantar a aplicação várias vezes ao longo do dia, será necessário reimplantar toda a aplicação para atualizar uma parte específica do código. Se estiver usando escalonamento em múltiplos servidores, serão várias reimplantações de toda a aplicação. Consequentemente, implantação contínua torna-se muito difícil de ser realizada [Richardson 2015].

Outro problema com aplicações monolíticas é que como módulos estão sendo executados no mesmo processo, um *bug* em um módulo pode derrubar toda a aplicação, ou seja, todos os seus módulos. A adoção de novos *frameworks* também torna-se um problema, pois se há muito código empregando um determinado *framework* e se deseja substituí-lo por outro, provavelmente, será necessário rescrever todo este código para utilizar o novo *framework* [Richardson 2015].

Para solucionar estes problemas, muitas empresas, tais como, Amazon, Netflix, The Guardian e outros estão usando arquitetura de microsserviços [Di Francesco et al. 2017], cuja ideia é dividir a aplicação em um conjunto de serviços menores interconectados. Um microsserviço tipicamente implementa um conjunto de funcionalidades do negócio. Cada microsserviço é uma "mini-aplicação", construída independentemente como ilustrado na Figura 1.7. Microsservicos expõem/consomem funcionalidades para/de outros microsserviços. Alguns microsserviços podem implementar interfaces web, ou seja, mesmo as interfaces web podem ser disponibilizadas como microsserviços independentes. Isto permite implantar experiências distintas para usuários ou dispositivos específicos ou para casos de uso específicos.

Alguns microsserviços são expostos para aplicativos móveis e outras aplicações através de *gateways*. Estes tem o objetivo de intermediar o acesso aos microsserviços, realizando tarefas como balanceamento de carga, *caching*, controle de acesso, medições monitoramento etc.

A fim de reduzir o acoplamento, cada microsserviço tem seu próprio banco de dados. Eventualmente, dados podem ter que ser replicados e tecnologias de bancos de dados distintas serem empregadas, como, por exemplo, um banco de dados que executa consultas espaciais com alto desempenho [Sadalage and Fowler 2012].

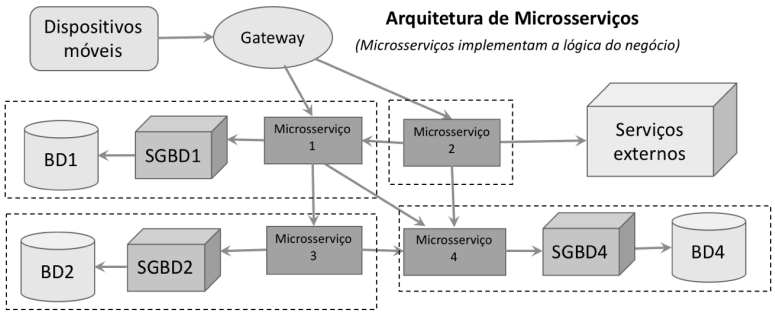



**Figura 1.7. Arquitetura empregando microsserviços (adaptado de [Richardson 2015])**

A arquitetura de microsserviços trata o problema da complexidade através da decomposição da aplicação monolítica em um conjunto de microsserviços. Cada microsserviço tem uma fronteira muito bem definida o que garante o nível de modularidade na prática, o que é muito difícil de alcançar em uma aplicação monolítica. Consequentemente, serviços individuais são mais fáceis de desenvolver, entender e manter. Times independentes trabalham em cada microsserviço empregando as tecnologias que consideram mais adequada (tanto de software como de hardware), de acordo com as regras da organização em que trabalham. Cada microsserviço é implantado independentemente, facilitando a atualização de novas versões sem impactar a implantação de outros microsserviços [Richardson 2015].

Não existe bala de prata e a arquitetura de microsserviços tem questões importantes a serem consideradas. Apesar de ser indicado que microsserviços sejam pequenos, o objetivo é que microsserviços correspondam a decomposições da aplicação a fim de facilitar desenvolvimento e implantação ágeis. Uma arquitetura de microsserviços corresponde a um sistema distribuído com comunicação interprocesso entre eles, o que é mais complexo do que invocar métodos no nível da linguagem, sendo também mais complexo de testar, por exemplo, necessidade de testes de integração entre os microsserviços. Particionamento do banco de dados entre os microsserviços traz o problema de garantia de integridade dos dados distribuídos entre eles. Existe uma maior quantidade de partes para serem configuradas, implantadas, escalonadas e monitoradas o que requer grande controle e alto nível de automação.

### 1.2.4.3. Princípios

Há sete princípios atribuídos a microsserviços [Zimmermann 2016, Fowler and Lewis 2014, Newman 2015]:

- **Interfaces com granularidade fina**: Unidades com responsabilidades específicas e independentes [Mehta and Heineman 2002] que encapsulam tanto lógica de pro-

cessamento quanto dados, onde o último é exposto através de APIs da Web ou filas de mensagens assíncronas.

- **Práticas de desenvolvimento orientadas ao negócio**: Técnicas e princípios para identificar e conceituar serviços como, por exemplo, o Projeto Orientado ao Domínio (Domain-Driven Desing - DDD) [Evans 2004]. Contexto bem definido (ou *Bounded Context*) é o padrão central do DDD [Vernon 2013]), o qual divide grandes domínios em contextos menores. Através do DDD equipes de desenvolvimento priorizam de forma sistemática os aspectos mais distintos e valiosos para a organização;

- **Princípios de design baseados em computação em nuvem**: Diretrizes como distribuição, elasticidade e baixo acoplamento[Fehling et al. 2014].

- **Entrega contínua descentralizada**: Promove um alto nível de automação e autonomia, demanda maturidade para construir artefatos de forma automatizada, aliados a uma suíte de testes [Humble and Farley 2010].

- **Contêineres leves**: Uso de metodologias de conteinerização de componentes de software para promover artefatos através de processos de entrega contínua (*e.g.*, Docker [Merkel 2014]).

- ***DevOps***: Uso de técnicas e automatização de configuração, desempenho e gerenciamento de falhas, estendendo práticas ágeis para o monitoramento de serviços [Hüttermann 2012].

- **Múltiplos paradigmas**: Combina ganhos de diferentes abordagens de computação e de tipos distintos de armazenamento[Wampler and Clark 2010].

#### 1.2.4.4. SOA x MSA

Alguns autores entendem a arquitetura de microserviços (Microservices Architecture - MSA) como uma abordagem mais específica de SOA ou como um padrão de SOA (isto é, "padrão de microsserviços") onde SOA preconizaria o uso de serviços grossos (isto é, com granularidade grossa) e microsserviços indica o desenvolvimento de serviços mais finos (isto é, granularidade fina) [Richardson 2016]. Outros apontam a arquitetura de microsserviços como sendo algo completamente novo.

Di Francesco *et al.* apresentam que MSA surge de SOA, mas apresenta diferenças importantes em relação ao segundo. MSA foca em aspectos específicos de SOA, tais como, componentização de pequenos serviços leves, uso de práticas ágeis e DevOps para desenvolvimento, utilização de automação de infraestrutura com entrega contínua, gestão de dados descentralizada e governança descentralizada entre serviços. Dentre as diferenças entre MSA e SOA tem-se: o fato do projeto de serviços em MSA ser direcionado a uma filosofia de compartilhamento de nada (*"share-nothing philosophy"* para o uso de métodos ágeis e promover isolamento e autonomia, enquanto que SOA busca alto grau de reuso (*share-as-much-as-you-can"*); MSA foca em coreografia enquanto SOA foca em orquestração e coreografia [Richards 2015] *appud* [Di Francesco et al. 2017].

#### 1.2.4.5. Vantagens e desvantagens

Resumindo, as vantagens de uma arquitetura de microsserviços são:

- Os microserviços são pequenos: (i) melhoram o isolamento de falhas; (ii) código é facilmente compreendido; (iii) tornam os desenvolvedores mais produtivos; (iv) iniciam muito mais rápido.
- Serviços individuais são mais fáceis de entender e podem ser desenvolvidos e implantados de forma independente. Cada microsserviço pode ser implantado em um hardware mais adequado para as exigências de seus recursos.
- Adotar novas tecnologias e *frameworks* torna-se mais fácil, pois a adoção pode ser aplicada em um microsserviço de cada vez.
- Cada microsserviço pode ser escalonado de forma independente através da duplicação e particionamento.
- A arquitetura de microserviços é adequada para aplicações complexas e de grande porte que estão evoluindo rapidamente como, por exemplo, Netflix e Spotfy.

Como desvantagens da arquitetura de microsserviços temos:

- As complexidades adicionais do desenvolvimento de sistemas distribuídos: (i) Aplicações muito mais complexas e constituídas por mais elementos; (ii) Uso de transações distribuídas ou consistência eventual; (iii) Gerenciamento de um número maior de aplicações em produção.
- Para ser utilizada de forma eficaz, a arquitetura de microsserviços exige um alto nível de automação.

### 1.2.5. Contêineres

Contêiner é um termo que descreve uma alternativa mais leve às máquinas virtuais. Para isso é feito o encapsulamento da aplicação em um ambiente virtual que contém apenas os ativos necessários para o funcionamento. Os contêineres são isolados a nível de disco, memória, processamento e rede. Essa separação permite uma grande flexibilidade, onde ambientes distintos podem coexistir na mesma máquina hospedeira (*host*), sem causar problemas.

Comparando contêiner com maquina virtual, temos que cada máquina virtual requer um Sistema Operacional próprio além dos softwares e bibliotecas. Além disso uma camada intermediária (chamada *hypervisor*) gerencia a comunicação de cada máquina virtual com o sistema operacional hospedeiro. Já os contêineres acessam diretamente o sistema sistema operacional hospedeiro e seus recursos (por exemplo, disco, memória, rede) para prover um ambiente virtual para as aplicações. Portanto, no ambiente de contêiner, é necessário instalar os requisitos (softwares, arquivos etc.) que o microsserviço precisa sem se preocupar com instalações de outro Sistema Operacional, além de não precisar do *hypervisor*.

### 1.2.6. Docker

Docker[9] é uma plataforma aberta criada utilizando o modelo de contêiner para "empacotar" a aplicação, que após ser transformada em uma imagem Docker, poderá ser reproduzida em plataforma de qualquer porte. O objetivo é facilitar o desenvolvimento, implantação e execução de aplicações em ambientes isolados da forma mais rápida possível. O Docker permite gerenciar a infraestrutura da aplicação. O que agiliza o processo de criação, manutenção e evolução.

O Dockerfile é um arquivo que descreve as instruções para criar uma imagem de contêiner Docker. Uma imagem sempre deve partir de uma imagem base. Portanto, o Dockerfile descreve de uma forma textual a diferença entre a imagem base e a imagem que se deseja criar, isto é, o Dockerfile contém a sequência de instruções necessárias para modificar a imagem base para que ela fique com as características desejadas.

A Figura 1.8 apresenta um exemplo de Dockerfile. A linha 2 corresponde ao nome e a versão da imagem base. A linha 4 contém informações do mantenedor da imagem base. A linha 6 define variáveis de ambiente. A linha 8 executa uma instrução na linha de comando. Neste caso, é executado o `apt-get`[10] para instalar a biblioteca `openjdk`[11]. A linha 10 executa o comando `ADD` que copia arquivo para diretório de trabalho (`WORKDIR`) do contêiner. Neste caso, está sendo copiado o `arquivo_test` da raiz para o diretório `temp` da raiz do `WORKDIR`.

```
1  # Container Docker
2  FROM ubuntu:14.04
3
4  MAINTAINER nome mantenedor <email@gmail.com>
5
6  ENV DEBIAN_FRONTEND noninteractive
7
8  RUN apt-get install openjdk
9
10 ADD ./arquivo_teste /tmp/
```

**Figura 1.8. Dockerfile transformando imagem base em imagem modificada.**

Docker Hub[12] provê um recurso centralizado para descoberta, distribuição e gestão de mudanças de imagens de contêineres, colaboração de usuários, e automação de *workflow* através de um *pipeline* de desenvolvimento.

Docker-Compose é a ferramenta do Docker que permite a criação e a execução de aplicações baseadas em múltiplos contêineres. Através dele, é possível construir as imagens dos contêineres e iniciar os serviços da composição. Cada contêiner mantém seu isolamento, mas é possível que um contêiner "identifique" outro através do nome do host usando o comando "link" - sem a necessidade de usar ip fixo. Também é possível criar **volumes**, que são pastas no Sistema Operacional hospedeiro onde podem ser armazenados

[9] `https://www.docker.com/`

[10] O apt-get é um recurso desenvolvido originalmente para a distribuição Debian que permite a instalação e a atualização de pacotes (programas, bibliotecas de funções, etc) no Linux.

[11] OpenJDK ("Open Java Development Kit") é uma implementação livre e gratuita da plataforma Java, Edição Standard ("Java SE").

[12] https://docs.docker.com/docker-hub/

os arquivos persistentes de aplicações executando no contêiner (como os de bancos de dados).

A configuração de uma composição de contêineres no Docker é descrita através do arquivo `docker-compose.yml`. Ele é um arquivo na sintaxe YAML[13] onde é possível descrever serviços, redes e volumes da composição.

A Figura 1.9 apresenta um exemplo de composição utilizando o Docker Composer. A linha 1 define o nome do contêiner. A Seção "links" listam os contêineres que serão compostos, neste caso, *db* e *redis* - linhas 3 e 4 respectivamente. A imagem *redis* tem o nome *redis* e a imagem *db* tem o nome "postgres" - linhas 7 e 9, respectivamente. Finalmente, a Seção "volumes" define os volumes dos contêineres. Neste caso, o contêiner *db* tem, na linha 11, o mapeamento do diretório `/opt/bancodados/` do Sistema Operacional hospedeiro mapeado no diretório `/usr/local/pgsql/data`.

```
1  web:
2    image: httpd
3   links:
4    - db
5    - redis
6  redis:
7    image: redis
8  db:
9    image: postgres
10   volumes:
11       - /opt/bancodados/:/usr/local/pgsql/data/
```

**Figura 1.9. Exemplo de composição usando Docker Composer.**

Algumas alternativas ao Docker são:

- **LXD ("Lex-Dee")**[14]: proposta de contêineres que operam como máquinas virtuais.
- **Rocket (rkt)**[15]: emprega o conceito de *pod* - uma coleção de uma ou mais aplicações executando em um contexto compartilhado. Configurações podem ser feitas a nível de *pod* ou a nível da aplicação. A arquitetura do rkt preconiza que cada *pod* executa diretamente em um modelo de processo Unix (isto é, não existe *daemon* central), em um ambiente auto-contido e isolado.

## 1.3. Persistência Poliglota em Microsserviços

Esta seção apresenta os conceitos de persistência poliglota e estratégias para tratar este desafio em microsserviços.

### 1.3.1. Definição

O termo "Persistência Poliglota" é usado para explicitar uma abordagem de persistência híbrida[Sadalage and Fowler 2012], que propicia a utilização dos melhores mecanismos projetados para armazenamento e recuperação das informações de acordo com a sua natureza. Isso é ilustrado no cenário apresentado na Figura 1.10, no qual uma plataforma de

[13] http://yaml.org/spec/
[14] `https://www.ubuntu.com/containers/lxd`
[15] `https://coreos.com/rkt/`

*e-commerce* consome quatro microsserviços, cada um com seu próprio SGBD (Sistema de Gerenciamento de Banco de Dados). Dados de um carrinho de compra (por exemplo, id de produto e quantidade) podem ser eficientemente resgatados a partir de um banco chave-valor (*key-value*), e ordens de compra podem ser persistidas de forma agregada e estruturada em um banco NoSQL Document, enquanto que um catálogo de produto pode ser mantido por meio de um banco relacional (SGBDR) e informações cruzadas de compra e clientes podem ser armazenadas em um grafo para inferir recomendações. Sendo assim, para atender diferentes requisitos tecnologias apropriadas de banco de dados são utilizadas.

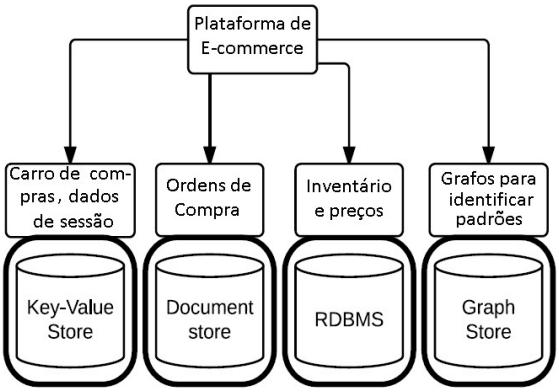



**Figura 1.10. Persistência Poliglota - Adaptada de: [Sadalage and Fowler 2012]**

### 1.3.2. Problemas

Descentralizar a responsabilidade pela manutenção de dados em microsserviços tem implicações no gerenciamento de atualizações dos dados e das consultas. Devido à complexidade na implementação de transações distribuídas, as arquiteturas de microsserviço enfatizam a coordenação em atualizações de serviço e delegações em consultas, de modo que a consistência ou a disponibilidade nem sempre serão garantidas. Isso é uma característica inerente a qualquer sistema distribuído que compartilha dados, conforme o estabelecido no teorema CAP [Brewer 2000] (*Consistency, Availability and Partition Tolerance*), pelo qual afirma-se que no máximo dois fatores - dentre tolerância a partição, consistência e disponibilidade - podem ser plenamente e simultaneamente atendidos. Dado que partição de rede é inerante em uma arquitetura de microsserviços, nós temos que balancear entre disponibilidade e consistência, os quais combinados com diferentes tecnologias de persistência trazem novas demandas e requerem integrações complexas [Fowler 2015].

### 1.3.3. Estratégias de Abordagem dos Problemas

Existem cinco estratégias para consultar e integrar informações distribuídas, em um cenário de persistência poliglota em uma arquitetura de microsserviço, com base em diferentes abordagens encontradas na literatura.

Um conjunto inicial de quatro estratégias foi definido de acordo com abordagens específicas para microservices [Newman 2015] (Seções 1.3.3.1, 1.3.3.2, 1.3.3.3,

1.3.3.4). Esta lista foi complementada por uma estratégia alternativa (Seção 1.3.3.5) baseada em estudos científicos [Ghawi and Cullot 2007, Collins et al. 2002] publicados antes de 2012. Na época, o termo "microservices"[Fowler and Lewis 2014] ainda não havia sido cunhado, e eles foram escolhidos devido à sua proposta e à similaridade de arquitetura. Nesta seção, apresentamos as estratégias bem como a avaliação das mesmas conforme Villaça *et al.* [Villaca et al. 2018], incluindo seus aspectos positivos e negativos mais relevantes. Artigos relacionados, alinhados a cada estratégia, também são apresentados. Villaça *et al.* complementam seu artigo com um quadro comparativo das estratégias conforme o seu grau de adequação à ISO 25010, uma norma de qualidade de software amplamente utilizada como referência.

#### 1.3.3.1. Shared Database

Essa estratégia sugere o aproveitamento de recursos de compartilhamento disponíveis nos próprios bancos de dados de forma que vários serviços possam se beneficiar da mesma fonte de dados [Newman 2015]. Três mecanismos podem ser usados nesta abordagem para gerir os dados [Messina et al. 2016]: (i) Cada serviço tem um conjunto de tabelas em um esquema compartilhado; (ii) Cada serviço tem um esquema em um SGBD compartilhado; (iii) Cada serviço tem seu próprio SGBD.

Essa última opção (Figura 1.11) pode usar diferentes tecnologias para armazenamentos de dados, demandando a utilização de soluções como *gateways*, conectores ou federações de dados, que podem implicar em custo de maior latência para recuperar informações.

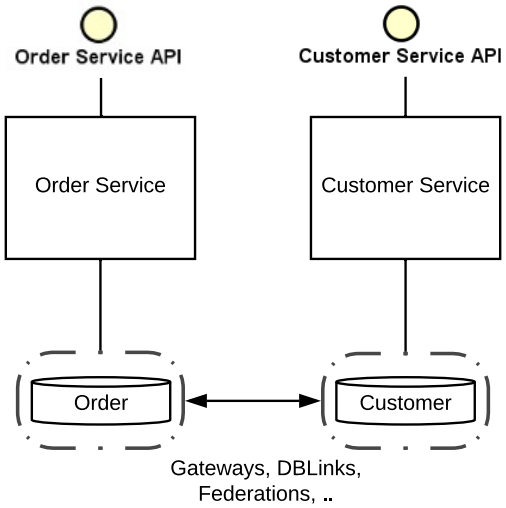



**Figura 1.11. Bancos de Dados compartilhados entre microsserviços**

Dentre seus aspectos positivos e negativos mais relevantes [Villaca et al. 2018] destacamos:

- **Prós**

1. Desenvolvimento simplificado, uma vez que cada serviço dispõe de acesso direto às fontes de informação;
2. Bons desempenho se o número de nós reduzido, dado que serviços podem efetuar filtros e junções de uma maneira otimizada. O tráfego de rede é reduzido devido à baixa necessidade de colaboração entre os serviços uma vez que o banco de dados é compartilhado.

- **Contras**

1. Alto Acoplamento. Mudanças em esquemas e indisponibilidade de banco de dados impactam todos os componentes que o utilizam. Com o incremento do número de serviços aumenta o risco de impacto das operações transacionais sobre as consultas, e vice-versa. A manutenibilidade dessa infraestrutura também é prejudicada, dado que as entregas (*releases*) dos serviços são vinculadas entre si. Além disso, a dependência a soluções de fornecedores afeta a portabilidade dos componentes envolvidos. Devido a esses aspectos, essa estratégia é vista como um anti-padrão para microsserviços [Richardson 2016].
2. Risco de baixa rastreabilidade. Atividades de teste e análise que dependem de avaliar o código executado demandam a existência de código aberto e compreensível pela equipe de desenvolvimento e operação.

**Exemplo prático:** Uma abordagem de implementação dessa estratégia considera que um banco de dados pode ser considerado um microsserviço por si só [Messina et al. 2016]. Uma vez que um banco de dados apresente uma arquitetura aberta e forneça mecanismos suficientes para estender seus recursos, ele pode incorporar a lógica de negócios que implementa o serviço desejado, atuando como um serviço de negócios. Para tal, Messina *et al.* usaram OrionDB[16], um banco de dados NoSQL multimodelo que suporta a persistência de modelos em grafo e como documentos, e permite o relacionamento entre os dados de ambos via API. Uma das vantagens dessa implementação é que em casos de muitas instâncias o cluster do banco atua como um registro de serviços (*service registry*).

#### 1.3.3.2. Command Query Responsibility Segregation (CQRS)

CQRS preconiza a utilização de uma visão materializada dos bancos de dados transacionais para fins de recuperação de informações de modo a isolar as operações de carga das operações de consulta [Newman 2015, Richardson 2016]. Nesta estratégia, os modelos devem ser divididos em duas partes (Figura 1.12): (i) *command-side*: responsável pela atualização de informações; (ii) *query-side*: responsável pela execução de consultas.

No *command-side* as operações de alterações (*i.e.*, inserções, atualizações e remoções) são validadas e, se aceitas, são aplicadas ao seu modelo. Seus componentes emitem eventos sempre que os dados são alterados, sinalizando alterações de estado. Esses eventos são publicados em uma fila ou armazenados em um meio como um banco de dados.

[16] `http://orion.cs.purdue.edu/`

O *query-side* consome o fluxo de eventos emitidos para capturar os dados alterados. Este módulo pode criar projeções a partir de eventos armazenados para montar o estado de objetos de domínio (processo conhecido como *event sourcing*), ou simplesmente atualizar um tipo diferente de armazenamento (como um *data warehouse*).

Essa separação permite diferentes tipos de escalonamento. As partes de comando e consulta podem ser implantadas como serviços separados, em infraestruturas de hardware separadas e com diferentes tipos de armazenamentos de dados [Newman 2015].

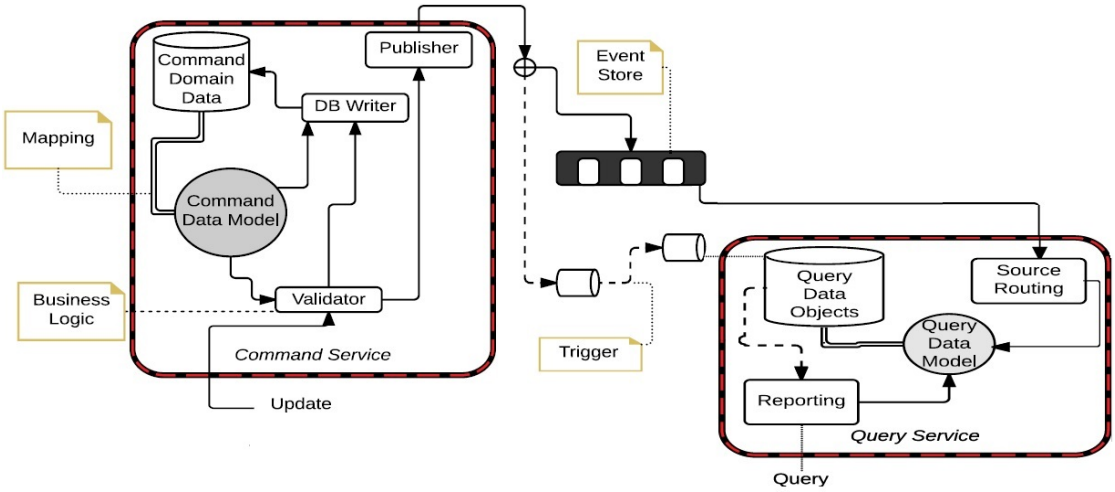



**Figura 1.12. *Command-side* e *Query-side* - CQRS**

- **Prós**

  1. Otimização de recursos e redução do tempo de resposta, especialmente em cenários em que buscas em grandes volumes de dados impactam operações transacionais;
  2. Serviços apresentam capacidade de evoluir de acordo com sua demanda, sem impactar os demais. Essa abordagem favorece o projeto de componentes mais coesos e fracamente acoplados.

- **Contras**

  1. Eventos de falha na sincronização, ou mesmo de longa duração para sua concretização, podem comprometer a consistência.
  2. Complexidade no desenvolvimento e operacionalização da arquitetura, necessidade de lidar com estratégias de tratamentos de eventos fora da ordem de ocorrência, de idempotência, etc.

**Exemplo prático:** ***Medical.Net*** [Rajkovic et al. 2013] é um sistema de informação que foi evoluído para seguir esta estratégia. Ele é dedicado para instalações médicas e agrega informações de formulários de admissão de pacientes, exames médicos, análises laboratoriais e tratamentos terapêuticos. A versão inicial deste sistema começou a mostrar

respostas lentas à medida que a quantidade de informações ficou maior, com dados espalhados em tabelas separadas. O desempenho das consultas foi afetado, devido ao aumento no tráfego de dados nas operações de junção (*join*). Para reduzir o impacto nas operações de carga no banco de dados principal, um componente separado foi criado apenas para leitura de dados, juntamente com um mecanismo de sincronização que apresenta o seguinte funcionamento: agentes (microsserviços) capturam alterações de dados específicas dos bancos transacionais (no *command-side*) e as replicam em bancos de dados (somente leitura) desnormalizados para otimizar consultas, usando estratégias para invalidar ou remover registros alterados e migrar novos dados.

#### 1.3.3.3. Event Data Pump

Event Data Pump é muito similar à estratégia CQRS, cuja definição é mais estrutural, essa estratégia apresenta uma definição comportamental. Atualizações de informações de serviços são capturadas e publicadas como uma série de eventos em uma plataforma (Figura 1.13) que permite seu consumo pelos demais serviços interessados [Newman 2015]. Esses serviços (receptores) são responsáveis por manter o estado atualizado de suas fontes de dados, com base nas informações obtidas nos eventos que são do seu interesse.

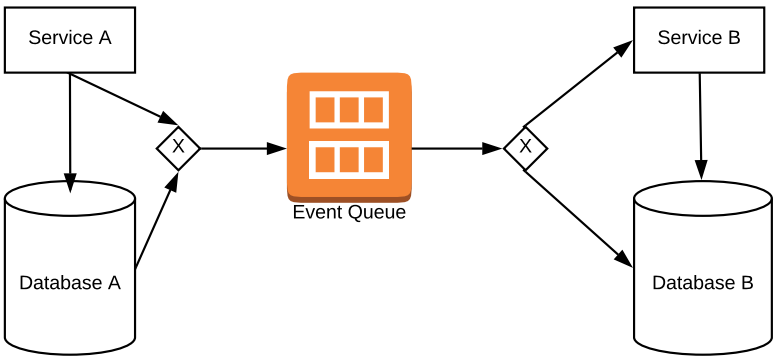



**Figura 1.13. Event Data Pump**

- **Prós**
  1. Ganho de desempenho, especialmente em cenários de cargas contínuas com um grande volume de dados, e necessidade de processamento e integração com outros dados em tempo próximo ao de ocorrência dos eventos;
  2. Serviços mais coesos e fracamente acoplados.
- **Contras**
  1. Falhas de um publicador ou receptor podem afetar a consistência das informações obtidas, e, devido à natureza distribuída da arquitetura, podem se tornar difíceis de investigar;
  2. Complexidade nas atividades de desenvolvimento e operação.

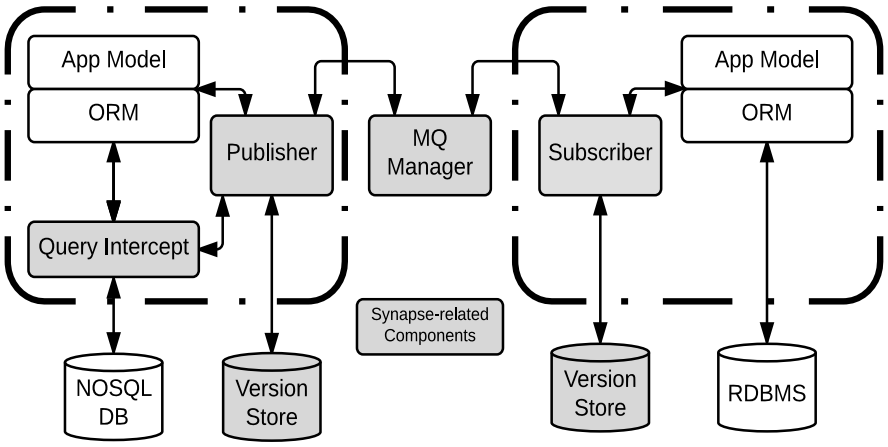



**Figura 1.14. Componentes Synapse - adaptado de [Viennot et al. 2015]**

**Exemplo prático:** Synapse [Viennot et al. 2015], um sistema de replicação escalável, permite que diferentes serviços, lidando com diferentes estruturas de dados, sejam desenvolvidos de forma independente. Cada um pode compartilhar subconjuntos de seus dados com os outros, e o Synapse sincroniza esses subconjuntos de dados com base nas informações mapeadas em cada serviço. Nos nós que estão atualizando dados, um agente Synapse é posicionado antes do *driver* de banco de dados, que lhe possibilita interceptar as atualizações de todos os modelos publicados antes de serem persistidos no banco de dados (Figura 1.14). Dessa forma, o sistema identifica quais objetos estão sendo gravados e os transmite para um Publicador, usando todos os atributos de objetos criados ou atualizados para criar uma mensagem de gravação. O Synapse envia a mensagem para um sistema de mensagem confiável, persistente e escalável, que a redistribui para seus serviços consumidores (assinantes).

#### 1.3.3.4. Data Retrieval via Service Call

A ideia dessa estratégia é extrair os dados necessários dos serviços de origem por meio de chamadas API (Figura 1.15). Para produzir relatórios reunindo dados de dois ou mais sistemas, são necessárias várias chamadas. No caso de precisarmos puxar grandes volumes de dados para compor uma visão de diferentes serviços, as consultas podem se tornar muito lentas; portanto, é preciso implementar estratégias para atenuar esse problema [Newman 2015].

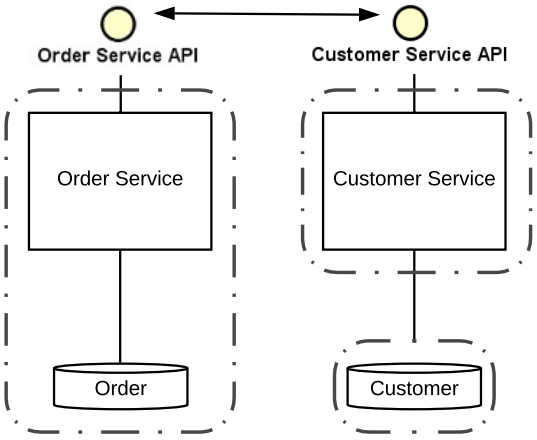



**Figura 1.15. Data Retrieval via Service Call**

- **Prós**

    1. Aplicações coesas podem ser reutilizadas e reagrupadas na composição de novas soluções;
    2. Novos serviços podem ser facilmente incorporados na infraestrutura.

- **Contras**

    1. APIs expostas podem não prever demandas de uso dos seus relatórios. O desempenho pode ser afetado ao executar operações que coletam dados de vários serviços (com diferentes meios de armazenamentos de dados). Além disso, redundância ou ambiguidade podem ser introduzidas se os serviços não forem cuidadosamente planejados, devido à autonomia.
    2. Cada componente deve ser capaz de tolerar falha, indisponibilidade ou atraso dos serviços que colaboram com o mesmo;
    3. As evoluções devem ser planejadas de maneira coletiva a fim de minimizar os impactos aos demais serviços.

**Exemplo prático:** Sheibel elaborou um caso de uso para este cenário [Scheibel 2016] no contexto de dados de trajetória de objetos móveis. A quantidade de dados manipulados por soluções nesta área é relevante - o tamanho da maior base de dados utilizada no estudo foi estimado em 820 GB de armazenamento. A fim de abstrair cada aplicação consumidora a partir de detalhes de quais microsserviços e informações do banco de dados devem ser utilizados, um novo serviço (chamado de Catálogo de Trajetórias) foi construído em uma camada acima de todos os microsserviços que lidam com repositórios. O mesmo coleta dados como informações de estado das trajetórias e de seus repositórios, propiciando o roteamento das solicitações para seus serviços específicos. Os resultados são tratados com base nos metadados de serviço mantidos por esse componente.

#### 1.3.3.5. Canonical Data Model

Um modelo de dados canônico (CDM) compreende a semântica e a estrutura da informação, de acordo com regras acordadas por várias partes [Gilpin 2015], simplificando a troca de dados entre os serviços. Essa estratégia é baseada no Serviço Central, que mantém um modelo universal, e de serviços subjacentes (serviços A e B) que mantêm modelos derivados do canônico, além de fontes de dados próprias. Esses serviços são capazes de mapear requisições, que chegam de acordo com seu modelo, aos dados estruturados em suas bases(Figura 1.16).

Para garantir compatibilidade entre todos os componentes, o Serviço Central abstrai os detalhes dos serviços que gerem as fontes de dados das visualizações dos usuários, de modo que traduz suas consultas em subconsultas (com base em modelos centrais e locais). As subconsultas são executadas nos serviços correspondentes, e, a partir disso, seus resultados são integrados para compor a resposta aos usuários.

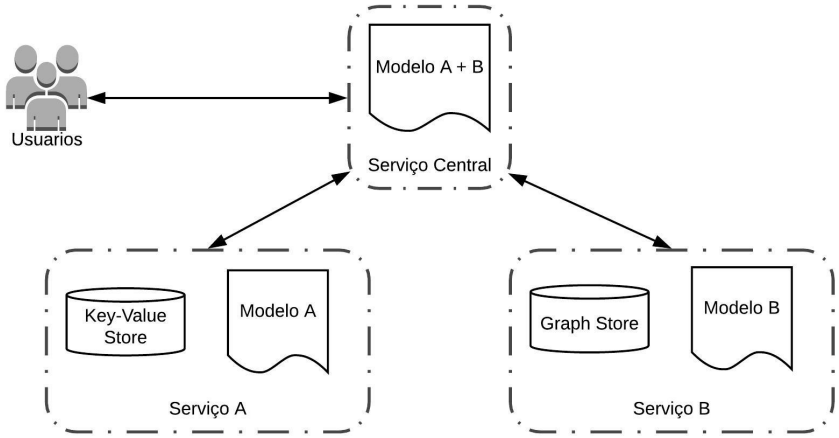



**Figura 1.16. Canonical Data Model**

- **Prós**

  1. Potencial para maior expressividade em pesquisas, especialmente para dados semânticos;
  2. Modelos canônicos podem auxiliar na validação de todos os modelos de dados dos serviços, uma vez que antecipam problemas de inconsistência e redundância dos dados;
  3. Potencial para composições de vários serviços e inferências de informações com base no modelo conhecido;

- **Contras**

  1. Complexidade na modelagem adequadamente as parcelas que compõem o CDM, na implementação da segregação do processamento, distribuído entre

nós delegados (que podem apresentar falhas), e na composição dos resultados conforme a requisição;

2. Dificuldade em manter um esquema canônico atualizado na medida em que as fontes distribuídas evoluem.

**Exemplo prático:** A proposta de Ghawi e Cullot [Ghawi and Cullot 2007] utiliza ontologias para a descrição semântica de fontes de informação usando uma ontologia híbrida - em que cada fonte de informação tem sua ontologia local, e uma ontologia de domínio global é usada para representar todo o domínio. Assim, cada ontologia relacionada à fonte de dados pode ser desenvolvida independentemente das ontologias de outras fontes.

## 1.4. Microsserviços na Prática

Esta seção apresenta práticas com microsserviços e persistência poliglota. A Seção 1.4.1 apresenta o cenário empregado na apresentação prática dos conceitos e também para ser utilizado no exercício prático deste trabalho. A Seção 1.4.2 apresenta a arquitetura tecnológica empregada. A Seção 1.4.3 apresenta os microsserviços de dados implementados. A Seção 1.4.4 apresenta os microsserviços de integração de dados, incluindo o microsserviço mediador implementado e o microsserviço de Processamento de Eventos proposto como exercício.

### 1.4.1. Cenário de Estudo

Um caso de uso foi elaborado a partir de um cenário de negócio presente em aplicações de *E-Commerce*. Embora bastante simples, contém elementos suficientes para apresentar uma quantidade relevante de desafios, vantagens e desvantagens usualmente observados no contexto de microsserviços. Ele foi construído a partir das estratégias de integração Data Retrieval via Service Call (Seção 1.3.3.4) e Event Data Pump (Seção 1.3.3.3) - neste caso foram propostos exercícios complementares.

O sistema possui como requisito apresentar informações relacionadas a carrinhos de compra dos clientes, tais como: produtos e quantidades, endereço de entrega cadastrado para o cliente e nome do fornecedor de um produto. Também deve apresentar informações isoladas de clientes e produtos para seus usuários. O diagrama ER (Figura 1.17) apresenta uma visão alto de nível do relacionamento entre as entidades envolvidas.

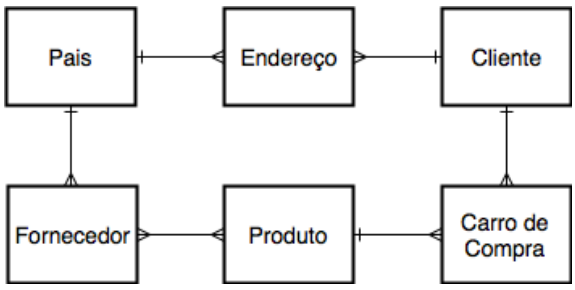



**Figura 1.17. Diagrama ER as entidades do negócio (elaborado empregando notação Engenharia de Informações [Heuser 2009])**

### 1.4.2. Arquitetura desenvolvida para o cenário

De acordo com o princípio de responsabilidade única de serviços (Seção 1.2.4.3), realizamos a divisão dos objetos como apresentado na Figura 1.18. Essa segregação proporciona maior grau de autonomia no desenvolvimento de serviços, dado que os mesmos só fornecem visibilidade de seus dados por meio de seu contrato (API) ou de mecanismos de replicação desacoplados de seus modelos de persistência. Cada parte desta divisão será disponibilizada por um microsserviço, o qual utilizará uma fonte de dados com tecnologia distinta dos demais, caracterizando um quadro de persistência poliglota.

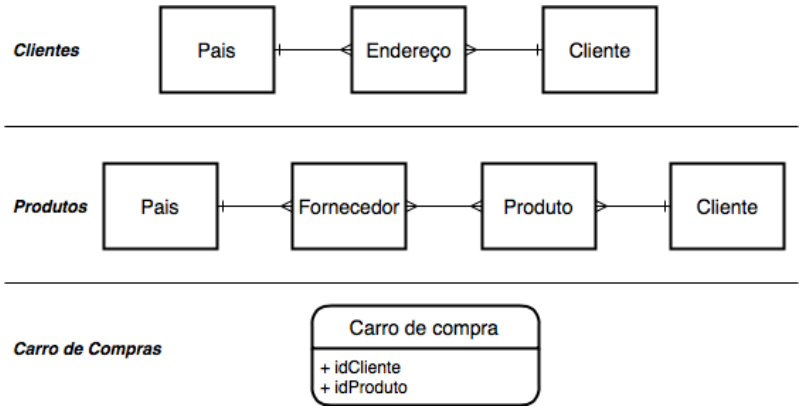



**Figura 1.18. Diagrama ER apresentando a divisão das entidades do negócio em domínios**

Além disso, outros componentes e serviços também comporão a solução a fim de prover mecanismos de integração, visualização e processamento de relatórios consolidado. A arquitetura dessa solução abrange os componentes apresentados na Figura 1.19. Os estereótipos de dada componente descrevem suas responsabilidades:

- Bancos de Dados: Fontes dos dados relacionados a cada divisão das entidades do negócio.
- Microsservicos de Dados: Responsáveis pelas atividades transacionais sobre os Bancos de Dados. Funcionam como uma camada de acesso a dados via API.
- Microsservicos de Integração: Consumidores dos dados providos via API ou Plataforma de Integração, são responsáveis por integrá-los e processá-los a fim de compor uma visão consolidada (ou mesmo ampliada) a partir das informações existentes. Este estereótipo foi aplicado a três componentes distintos:
  - Mediador: Utiliza estratégia Data Retrieval Via Service Call, consumindo dados via API dos Microsserviços de Dados e Processador de eventos.
  - Processador de eventos: via Plataforma de Integração, usa a estratégia Event Data Pump, por meio de mensagens subscritas a partir dessa plataforma. Ele atualiza uma fonte de dados própria, consolidando e agregando novas informações (como métricas estatísticas) requisitadas em relatórios.

– Plataforma de Integração: Componente que mantém os eventos publicados, a cada atualização das entidades de negócio, pelos Microsserviços de dados.

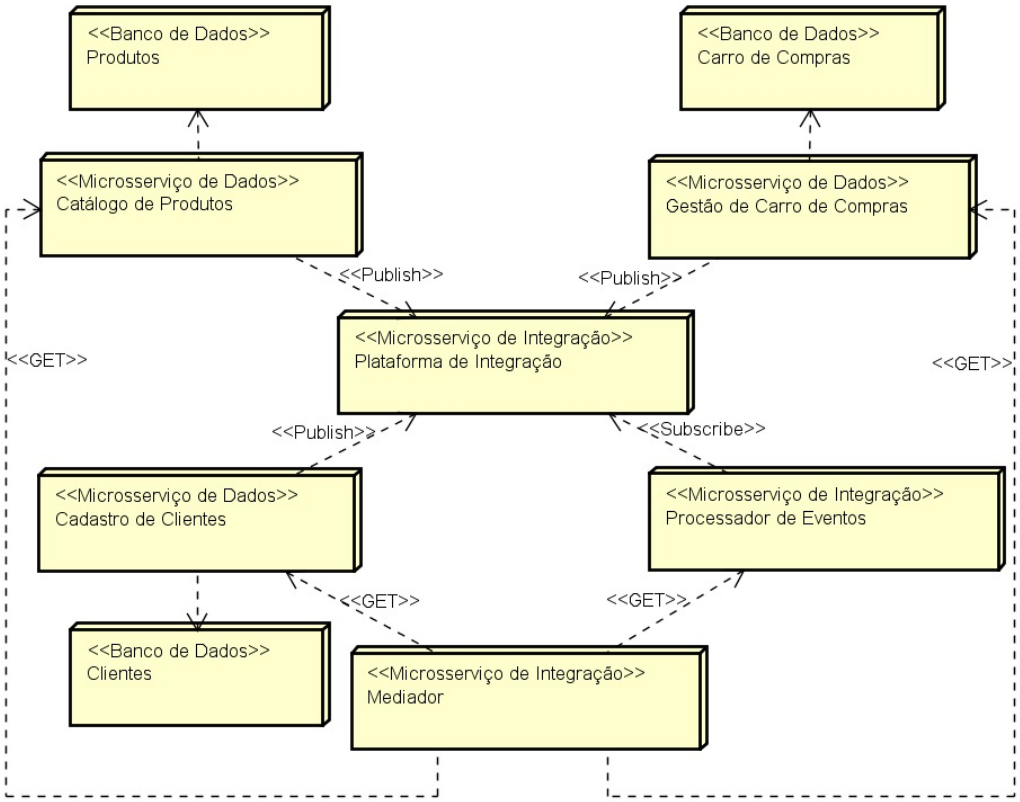



**Figura 1.19. Diagrama de Deployment - Arquitetura proposta**

#### 1.4.2.1. Repositório Git do trabalho

Um repositório Git[17] público foi criado no Bitbucket[18] para este trabalho disponibilizado na seguinte URL[19], contendo: (i) *readme* explidando o repositório; (ii) A estrutura de arquivos com o código fonte da arquitetura e dos exercício; (iii) Apresentação (*slides*) do trabalho; (iv) Outros links e material de apoio.

#### 1.4.2.2. Componentes tecnológicos comuns aos microsserviços

Três componentes tecnológicos foram utilizados para a criação dos microsserviços desse estudo.

[17] `https://git-scm.com/`
[18] `https://git-scm.com/`
[19] `https://luis-villaca@bitbucket.org/luis-villaca/tutorial_2018_sbsi.git`

**Gradle**[20]: *script* de *build* utilizado para empacotar aplicações e componentes web, traz uma série de facilidades como definição de atividade e mapeamento de dependências de bibliotecas via Maven[21]. O script Gradle é ainda utilizado para a execução das classes de teste e de eventual geração de métricas (como verificação do percentual de cobertura de código), build do artefato (caso o passo anterior não apresente erros) e posterior criação da imagem e contêiner Docker.

**SpringBoot**[22]: Permite a criação de aplicações autocontidas (com servidores web, aplicação e mesmo com bancos de dados), propicia integração com ferramentas que provêm métricas de qualidade de software, permite a configuração da aplicação Gradle. Essa tecnologia facilita a automatização na construção de pequenos serviços coesos e desacoplados.

**Java8**[23]: Dentre outras vantagens para versões anteriores, permite o processamento paralelo ou sequencial de dados em uma coleção (via StreamAPI[24]), com uma sintaxe mais clara e com maior desempenho do que a tradicional.

O site Spring IO provê um passo-a-passo[25] para se criar uma imagem Docker com SpringBoot e Gradle. Similar ao demonstrado no passo-a-passo, utilizaremos a estrutura de pastas indicada na Figura 1.20 para a confecção dos artefatos com SpringBoot (através do Gradle). Em determinados microsserviços essa configuração terá alguns pequenos ajustes.

#### 1.4.2.3. Bancos de Dados

Três tecnologias com modelos distintos de persistência foram utilizadas para os cenários segregados e são apresentadas a seguir. A maior motivação é demonstrar o cenário de persistência poliglota, mas outros fatores, como flexibilidade para a manutenção de estrutura dos dados, desempenho e necessidade de garantia de integridade nas operações transacionais devem ser considerados na escolha das tecnologias.

**DB I - Catálogo de Produtos**

Para este dominínio foi empregado o MongoDB[26] um banco NoSQL Document, que provê índices, capacidade de consultas simples e uma estrutura de dados mapeada em um esquema de dados flexível. Esta flexibilidade na estrutura de dados é importante porque os atributos de produtos podem mudar ao longo do tempo e se deseja que não haja grandes impactos devido a estas mudanças.

[20] https://gradle.org/
[21] https://maven.apache.org/
[22] https://projects.spring.io/spring-boot/
[23] http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html
[24] http://www.oracle.com/technetwork/pt/articles/java/streams-api-java-8-3410098-ptb.html
[25] https://spring.io/guides/gs/spring-boot-docker/
[26] https://www.mongodb.com/

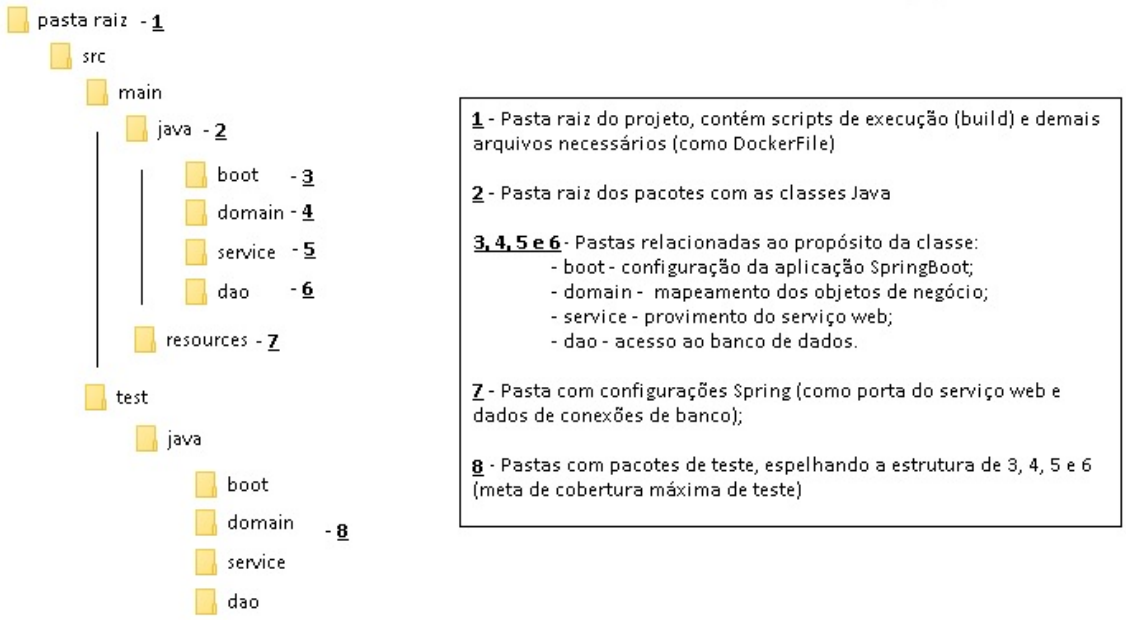



**Figura 1.20. Estrutura de pastas (Microsserviços)**

**Montagem deste ambiente**: A estrutura de contêiner (Seção 1.2.5) foi utilizada para facilitar a gestão da infraestrutura de cada componente, bem como prover uma configuração que permitisse que todos os componentes pudessem acessar, a partir de seus ambiente, os demais contêineres pelo atributo *name* registrado no Docker no momento de sua criação. Para tal, foi utilizado o Comando 1 para criar uma rede bridge, a qual será utilizada por este e pelos demais componentes. Nesse caso a execução sem parâmetros já instancia o servidor.

**Comando 1** Criando rede bridge

```
sudo docker network create -d bridge grupo_microsservicos
```

Para a criação do ambiente para o Mongo, é necessário baixar uma imagem padrão e executar o comando de criação do contêiner na rede criada (Comando 2).

Para confirmar a execução basta criar uma sessão iterativa rodando o Comando 3. Seguido de *db* (verifica base corrente) ou qualquer outro comando.

## DB II - Carro de Compras

Para este domínio, optou-se pelo Redis[27], um banco NoSQL Chave-Valor, opção escolhida devido ao acesso aos dados ser predominantemente via chave primária. A API utilizada para o consumo de dados do Redis também permite a busca em valores em dados serializados.

[27] `https://redis.io/`

**Comando 2** Comandos para baixar imagem padrão do MongoDB e criar container na rede

```
docker pull mongo:3.7-jessie
```

```
docker run --name mongodb -p 27017:27017 -d
--network grupo_microsservicos mongo:3.7-jessie
```

*Parâmetros indicam:*
`name`: *nome da imagem(redisdb)*;
`p <porta externa>:<porta interna>`: *expõe a porta rodando no contêiner na porta da máquina hospedeira*
`d`: *o modo de execução detached*
`network grupo_microsservicos`: *rede da qual o contêiner fará parte*
`mongo:3.7-jessie`: *nome da imagem docker baixada na máquina hospedeira*

**Comando 3** Iniciar seção no mongo

```
docker exec -it mongodb mongo
```

**Montagem** Para a criação do banco executar Comando 4 para baixar uma imagem padrão do redes e executar o comando de criação do contêiner. Para confirmar a execução basta rodar o Comando 5.

**Comando 4** Comandos para baixar imagem padrão do redis e criar container na rede

```
docker pull redis:3.2.11
```

```
docker run --name redisdb -p 6379:6379 -d --network
grupo_microsservicos redis:3.2.11 redis-server
--appendonly yes
```

*Parâmetros semelhantes ao do Comando 2:*
`redis-server -appendonly yes`: *comando para instanciar o servidor*

**DB III - Cadastro de Clientes**

Para este domínio, optou-se pelo SGBD MySQL[28], que provê garantias de atomicidade, consistência, isolamento e durabilidade (ACID) nas suas transações.

**Montagem** Para a criação do banco, executar Comando 6 para baixar uma imagem padrão e criar contêiner na rede criada. observe a senha gerada usando `docker logs mysqldb`. Conecte ao MySQL, crie um banco de dados e um usuário para a aplicação, executando Comando 7.

[28] `https://www.mysql.com/`

**Comando 5** Iniciar sessão iterativa no redis

```
docker run -it --network grupo_microsservicos
--rm redis:3.2.11 redis-cli -h redisdb -p 6379
```

*Parâmetros*:
`it`: *roda no modo iterativo*;
`network grupo_microsservicos`: *rede da qual o contêiner fará parte*
`rm`: *máquina "efêmera"- após a execução ou em eventual parada o contêiner é removido*
`redis:3.2.11`: *nome da imagem docker*
`redis-cli -h redisdb -p 6379`: *comando para instanciar cliente*
*Seguido de `KEYS *` (verifica chaves no servidor) ou qualquer outro comando*

**Comando 6** Comandos para baixar imagem padrão do MySQL e criar container na rede

```
docker pull mysql/sqlserver:5.7.22
```

```
docker run --name mysqldb -p 3306:3306 -d
--network grupo_microsservicos mysql/mysql-server:5.7.22
```

### 1.4.3. Microsserviços de Dados

Os microsserviços de dados formam a base dessa arquitetura: são os gerenciadores dos dados do negócio e proveem uma API Rest (Seção 1.2.2.2) aos demais serviços interessados em consumir informações disponibilizadas por eles. É necessário que seus bancos de dados já estejam ativos e configurados na mesma rede Docker em que serão instanciados os seus contêineres (Seção 1.4.2.3). O repositório indicado na Seção 1.4.2.1 contém todos os arquivos necessários a geração desse componente.

#### MS I - Catálogo de Produtos

Esse microsserviço é responsável pelo acesso ao banco de dados de Catálogo de Produtos, incluindo informações de Fornecedor. Este serviço foi desenvolvido empregando Spring Boot e acesso ao MongoDB[29].

O script Gradle para este microsserviço inclui[30]:

- Dependências para a execução do próprio script: Inclui a referência para a busca das bibliotecas e são indicadas as dependências para a execução do script.
- Configuração do artefato, da imagem e do contêiner Docker: Indica o nome do artefato (jar) gerado e faz referência ao Docker file (que estará na pasta raiz) a ser utilizado no processo de build. Além disso, inclui instruções para o posicionamento desse arquivo a fim de que seja devidamente referenciado no Dockerfile.

[29] O site Spring IO provê um guia `https://spring.io/guides/gs/accessing-data-mongodb/` para demonstrar a criação de um ambiente Spring Boot com MongoDB

[30] `https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-produtos-script-gradle`

**Comando 7** Comandos para criar banco de dados e usuário no MySQL

```
docker exec -it mysqldb mysql -uroot -p<senha gerada>

create database clientdt
create user 'usuariocli'@'%' identified by 'senhacli';
grant all on *.* to 'usuariocli'@'%';
```

- Configuração das dependências do código da aplicação: Inclui a referência para a busca das bibliotecas dependentes, indicando as versões da JVM e dependências para a execução do script.

Além desse script, o YAML da aplicação[31], localizado na pasta resources (Figura 1.20), indica três configurações: a porta utilizada para o Tomcat embutido no SpringBoot; as configurações de conexão com o banco de dados; e, as configurações do log4j (biblioteca de log) - relacionando os pacotes, o padrão das linhas geradas e a saída padrão do contêiner como destino.

A configuração do Docker para o microsserviço Produtos[32] indica: a imagem a ser utilizada (Linux Alpine com openjdk); a inclusão do arquivo jar (gerado pelo Gradle); e a execução do jar assim que o contêiner for executado.

A partir desta configuração, basta executar os comandos apresentados em Comando 8 para a geração da imagem Docker e execução do contêiner.

**Comando 8** Comandos para gerar microsserviço de catálogo de produtos

```
sudo ./gradlew.sh build buildDocker

sudo docker run --name ms01_catalogoprod
--network grupo_microsservicos --rm -p 8081:8080
class_unirio/catalogoprod_microservice:latest
```

O diagrama apresentado na Figura 1.21 indica os principais componentes e suas respectivas dependências, os quais são descritos a seguir de acordo com os estereótipos a eles atribuídos[33].

- **Boot**: Contém a classe principal do SpringBoot, codificada apenas com a instância da aplicação e com referências aos pacotes que auxiliarão no processamento das operações providas pelo serviço.

[31] https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-produtos-script-application-yml

[32] https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-produtos-docker-configuration

[33] https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-produtos-classes

- **Domain**: Contém as classes de domínio. Elas apresentam um quantidade de código reduzida, facilitando o desenvolvimento e a legibilidade do código. O componente Lombok foi utilizado nas anotações a fim de proporcionar esse ganho. Além disso, anotações do Spring Data MongoDB também foram utilizadas para mapear os atributos da classe com a estrutura Document persistida no banco.
- **DAO**: Contém a classe que provê acesso aos dados dos objetos de domínio, a qual também apresenta um quantidade de código reduzida, facilitando o desenvolvimento e a legibilidade.
- **Service**: Contém a classe que implementa as chamadas REST empregando o Spring Data REST.

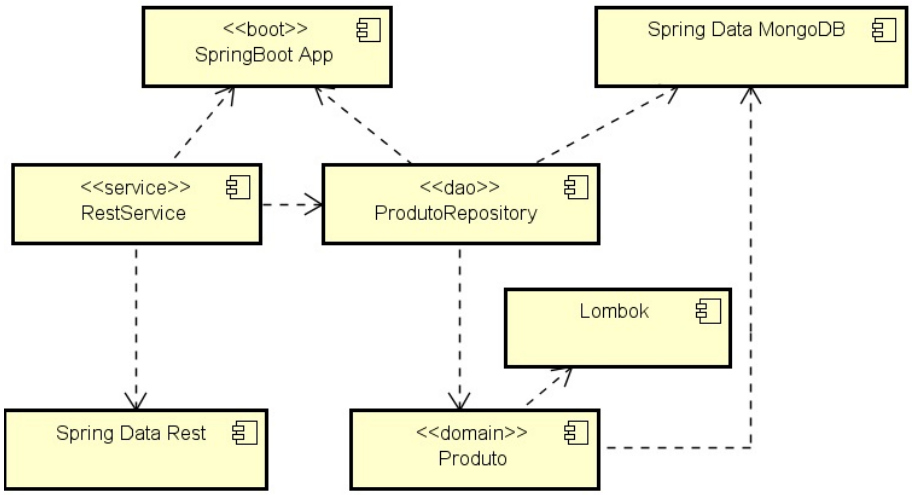



**Figura 1.21. Microsserviço de Dados - Catálogo de Produtos (Diagrama de Componentes)**

## MS II - Carro de Compras

Esse microsserviço é responsável pelo acesso ao banco de dados de Carros de Compra, que contém referências ao campo identificador do Cliente e a campos identificadores dos produtos escolhidos. Em especial, para este serviço, foi necessário configurar um artefato com Spring Data e Redis[34].

Os scripts utilizados para o microsserviço de Carro de Compras[35] são similares aos scripts empregados na construção do microsserviço de Produtos.

O diagrama apresentado na Figura 1.22 indica os principais componentes e suas respectivas dependências, os quais são descritos a seguir de acordo com os estereótipos a eles atribuídos[36].

[34] O site Baeldung provê um guia `http://www.baeldung.com/spring-data-redis-tutorial` para se configurar um artefato com Spring Data e Redis.

[35] `https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-carrodecompras-scripts`

[36] `https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-carrosdecompras-classes`

- **Boot**: Contém a classe principal do SpringBoot, codificada apenas com a instância da aplicação e com referências aos pacotes que auxiliarão no processamento das operações providas pelo serviço.
- **Domain**: Assim como no Microsserviço Catálogo de Produtos, as classes de domínio apresentam um quantidade de código reduzida, com anotações Lombok e Spring Data Redis para indicar o Hash a ser utilizado e mapear os atributos do objeto a ser serializado/desserializado.
- **DAO**: A classe CarroDeCompraRepository provê acesso aos dados dos objetos de domínio e apresenta as anotações Spring Data para o provimento das funcionalidades CRUD.
- **Service**: Classe que implementa as chamadas REST (assim como no Microsserviço Catálogo de Produtos). Neste caso, ela provê um serviço de busca sem filtro.

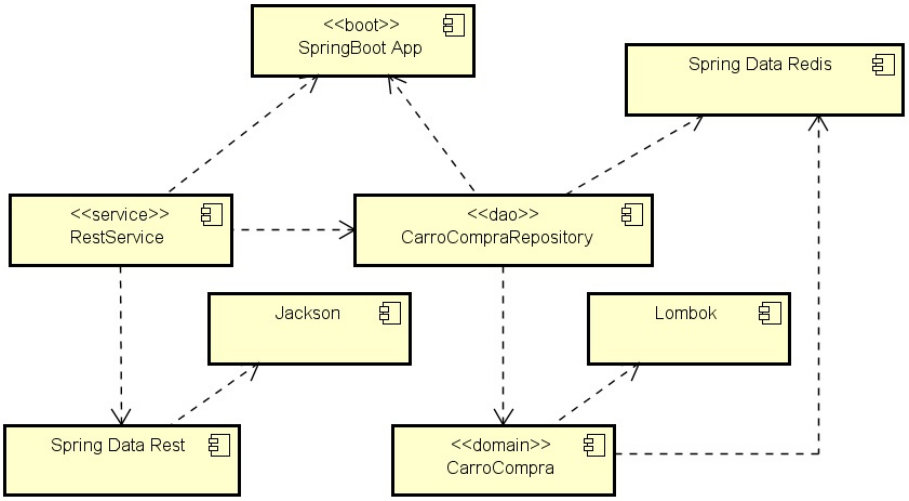



**Figura 1.22. Microsserviço de Dados - Carro de Compras (Diagrama de Componentes)**

**MS III - Cadastro de Clientes**

Esse microsserviço é responsável pelo acesso ao banco de dados de Cliente incluindo dados de seu endereço, que são armazenados em uma tabela própria no banco relacional. Este serviço foi desenvolvido empregando Spring Boot e acesso ao MySQL[37].

Os scripts utilizado para o microsserviço de Clientes é similar aos scripts para o microsserviço de Produtos, excetuando-se algumas dependências[38].

O diagrama apresentado na Figura 1.23 indica os principais componentes e suas respectivas dependências, os quais são descritos a seguir de acordo com os estereótipos atribuídos a eles.

[37] O site Spring IO provê um passo-a-passo`https://spring.io/guides/gs/accessing-data-mysql/` para se criar um artefato Spring Boot com MySQL.

[38] `https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-clientes-scripts`

- **Boot**: Contém a classe principal do SpringBoot, codificada apenas com a instância da aplicação e com referências aos pacotes que auxiliarão no processamento das operações providas pelo serviço.
- **Domain**: Assim como no Microsserviço Catálogo de Produtos, as classes de domínio apresentam um quantidade de código reduzida, com anotações Lombok e nesse caso com Spring Data JPA para mapear os atributos do objeto a ser persistido.
- **DAO**: Contém a classe que provê acesso aos dados dos objetos de domínio. Anotações Spring Data foram usadas para o provimento das funcionalidades CRUD.
- **Service**: Classe que implementa as chamadas REST (assim como no Microsserviço Catálogo de Produtos). Neste caso, ela provê um serviço de busca com filtro customizado.

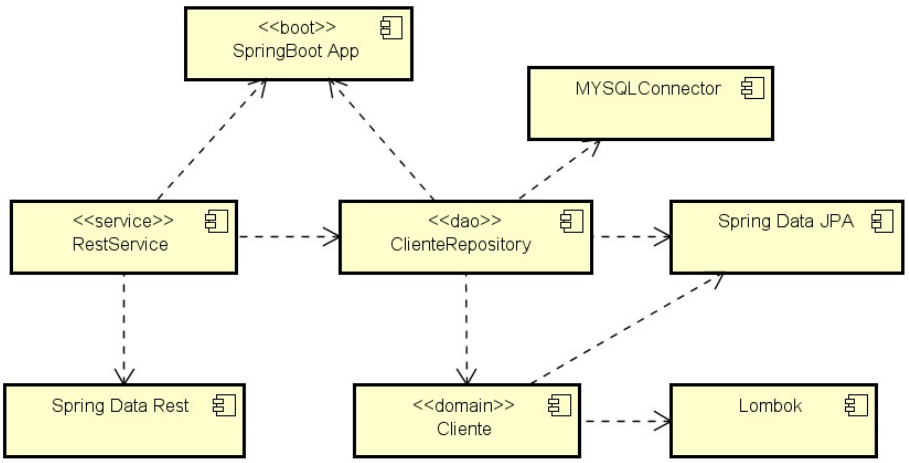



**Figura 1.23. Microsserviço de Dados - Cadastro de Clientes (Diagrama de Componentes)**

### 1.4.4. Microsserviços de Integração

Esses serviços atuam como consumidores dos microsserviços de dados, mediante a API disponibilizada ou outro mecanismo. São responsáveis por integrá-los e processá-los a fim de compor uma visão consolidada (no caso do serviço Mediador) ou ampliada (por meio do Processador de Eventos) a partir das informações existentes.

#### MS Integração I - Mediador

Esse microsserviço é responsável por prover uma visão GraphQL (Sessão 1.2.3) a partir da API web dos Microsserviços de Dados. Nesse caso não há acesso direto a uma base de dados. Os acessos são feitos via requisições REST aos microsserviços de dados, e as instâncias das classes de negócios são mapeadas a partir de suas respostas. Utiliza-se um Schema Graphql[39] para mapear estruturas de dados dessas instâncias e assim compor respostas às requisições.

[39] `https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-mediador-esquema-graphql`

Os scripts utilizados para o microsserviço Mediador[40] são similares aos scripts empregados na construção do microsserviço de Produtos.

O diagrama apresentado na Figura 1.24 indica os principais componentes e suas respectivas dependências os quais são descritos a seguir de acordo com os estereótipos a eles atribuídos[41].

- **Boot**: Contém a classe principal do SpringBoot, codificada apenas com a instância da aplicação e com referências aos pacotes que auxiliarão no processamento das operações providas pelo serviço.

- **Domain**: As classes de domínio são equivalentes às já apresentadas para os microsserviços de Produtos, Clientes e Carros de Compra, mas sem a necessidade de anotações que mapeiam informações de persistência dos atributos.

- **DAO**: Classes que proveem acesso aos dados dos objetos de domínio atuam, nesse caso, se conectando a um microsserviço de dados via chamadas REST.

- **DataFetcher**: Componentes chamados pelo mediador para a obtenção dos dados.

- **Service**: Corresponde a classe QueryMediator que é a classe mediadora que disponibiliza a API GraphQL.

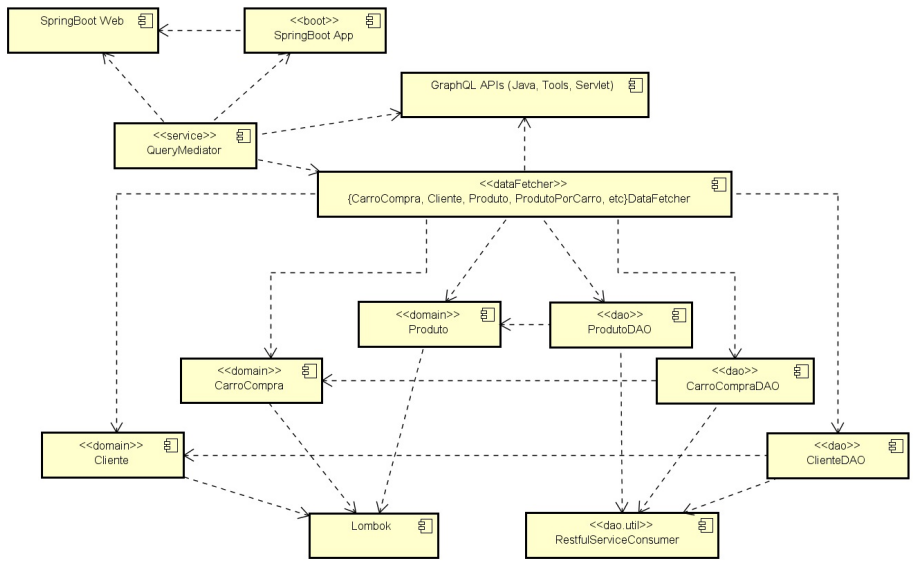



**Figura 1.24. Microsserviço de Integração - Mediador (Diagrama de Componentes)**

[40] https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-mediador-scripts

[41] https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-mediador-classes

**MS Integração II - Processador de Eventos**

Esse serviço é proposto como um exercício a fim de que sejam praticados os conceitos apresentados. Foi concebido a partir de um cenário em que surge um novo requisito: *trazer a média mensal de compras por país nos últimos 12 meses*.

Na atual arquitetura há algumas limitações que devem ser consideradas:

- O microsserviço de Carro de Compras não provê um mecanismo de busca por data, nem de busca ordenada. Logo, para utilizá-la todos os registros devem ser buscados e suas datas devem ser testadas.
- Alterar o microsserviço de Carro de Compras para prover uma busca por data não é suficiente, uma vez que ele não possui informações de clientes, de modo que uma nova busca seria necessária pelo serviço mediador para complementar o filtro.
- O microsserviço de Cadastro de Clientes possui o método *buscaPorNacionalidade*. No entanto, podem ser retornados muitos clientes. Logo, dependendo do número de clientes, da velocidade de processamento e da latência, pode-se optar por trazer todos as instâncias, manter um cache e sincronizar de tempos em tempos - ou, a cada registro de Carro de Compra aplicável, buscar o cliente (pelo seu id) o que também pode demandar muito tempo.
- Pode-se chegar à conclusão de que o mediador está onerado e que adicionar a estratégia Event Data Pump à arquitetura seja a melhor solução e, dessa forma, ter uma solução que permitirá que novos requisitos de integração para o cenário de Relatório sejam tratados com mais eficiência.

A proposta do exercício é utilizar uma plataforma de fila (sugerimos a utilização da RabbitMQ) para coletar os eventos de alteração dos dados dos microsserviços de dados, e prover um novo serviço de integração que será responsável por manter uma visão materializada de informações de compras e países (ou de mais informações se necessário). Esse serviço tem a atribuição de agregar dados continuamente, a partir dos eventos de alteração, e de gerar as estatísticas de forma contínua[42].

Nesse caso todos os serviços deverão contribuir para essa solução. Os microsserviços de dados deverão publicar suas alterações, e mesmo o nó Mediador pode ajustar o seu schema GraphQL com dados de Relatório (com, por exemplo, as médias mensais de valores de carros de compra por país), devendo para isso consumir também os dados gerados pelo nó Processador de Eventos, por meio de sua API.

## 1.5. Considerações Finais

O objetivo deste trabalho foi prover uma visão teórica da arquitetura de microsserviços e um guia prático para o desenvolvimento de artefatos nessa abordagem de arquitetura de modo a explicitar benefícios e desafios dessa abordagem em cenários específicos.

[42]Uma proposta de arquitetura para este novo serviço é apresentado em `https://bitbucket.org/luis-villaca/tutorial_2018_sbsi/wiki/ms-processadordeeventos-arquitetura-de-solucao`

Inicialmente foram apresentados os conceitos fundamentais das principais tecnologias relacionadas a microsserviços, seguidos dos desafios observados num contexto em que há necessidade de se relacionar dados a partir de fontes de informação distintas e distribuídas. Foram apresentadas as estratégias atualmente publicadas para a resolução desses desafios, destacando seus aspectos positivos e negativos.

Um cenário para a implementação de duas dessas estratégias foi elaborado para demonstrar suas características:

- Data Retrieval Via Service Call: Foi implementado o microsserviço Mediador, o qual consome informações de outros microsserviços via API Web para prover uma visão consolidada de dados;

- Event Data Pump: A implementação do microsserviço Processador de Eventos foi proposta como exercício a fim de praticar os conceitos apresentados na construção do microsserviço Mediador e de praticar o uso da estratégia Event Dasta Pump. O microsserviço Processador de Eventos subscreve e consome os eventos publicados pelos outros microsserviços e mantém sua própria visão materializada dos dados relevantes para seu contexto de processamento.

Nesse trabalho, foram exercitados cinco dos sete princípios de microsserviços (Seção1.2.4.3):

- Interfaces com granularidade fina: Os microsserviços de dados correspondem a unidades com responsabilidades específicas e independentes encapsulando tanto lógica de processamento como lógica de dados.

- Práticas de desenvolvimento orientadas ao negócio: Uso de DDD na segregação dos domínios para definição dos microsserviços de dados;

- Entrega contínua descentralizada: Várias tecnologias para automação foram empregadas na construção e implantação dos componentes;

- Contêineres leves: Utilização e configuração de imagens e contêineres Docker nas arquiteturas desenvolvidas;

- Múltiplos paradigmas: Foram utilizadas três tecnologias de bancos de dados (isto é, relacional (MySQL), orientada a documentos (MongoDB) e chave-valor (Redis)) para persistência bem como todas as tecnologias necessárias para construir os microsserviços que disponibilizavam acesso aos dados providos pelas mesmas.

O cenário proposto foi composto por nove componentes. A partir deles foi demonstrado algumas características buscadas em microsserviços, como pouco acoplamento e alta coesão, facilitando o desenvolvimento individual dos componentes.

A integração via API, na estratégia Data Retrieval Via Service, apresenta limitações, como a dificuldade de se estimar quão abrangente deve ser a exposição dos serviços.

O tratamento das operações que relacionam informações de diferentes bases de dados, cujos serviços estão distribuídos na rede, pode demandar uma abordagem diferenciada - como o caso da estratégia Event Data Pump.

A integração via Event Data Pump apresenta pontos positivos e negativos, assim como a estratégia Data Retrieval Via Service Call, conforme apontado na Seção1.3.3.

Portanto, por meio da discussão dos conceitos de microsserviços e de sua aplicação num cenário de persistência poliglota foi possível apresentar as alternativas que visam resolver aspectos importantes na construção de sistemas distribuídos e para integração de dados.

## Referências


[Booths et al. 2004] Booths, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. (2004). Web services architecture. `http://www.w3.org/TR/ws-arch/`. Acessado em 06/05/2018.

[Brewer 2000] Brewer, E. A. (2000). Towards robust distributed systems. In *PODC*, volume 7.

[Collins et al. 2002] Collins, S. R., Navathe, S., and Mark, L. (2002). Xml schema mappings for heterogeneous database access. *Information and Software Technology*, 44(4):251–257.

[Di Francesco et al. 2017] Di Francesco, P., Malavolta, I., and Lago, P. (2017). Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 21–30. IEEE.

[Erl 2005] Erl, T. (2005). *Service-oriented architecture: concepts, technology, and design*. Pearson Education India.

[Evans 2004] Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[Fehling et al. 2014] Fehling, C., Leymann, F., Retter, R., Schupeck, W., and Arbitter, P. (2014). *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer Science & Business Media.

[Fielding 2000] Fielding, R. T. (2000). Rest: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*.

[Fowler 2015] Fowler, M. (2015). Polyglot persistence. 2015. `https://martinfowler.com/bliki/PolyglotPersistence.html`. Acessado em 04/05/2018.

[Fowler and Lewis 2014] Fowler, M. and Lewis, J. (2014). Microservices. https://martinfowler.com/articles/microservices.html. Acessado em 02/02/2018.

[Ghawi and Cullot 2007] Ghawi, R. and Cullot, N. (2007). Database-to-ontology mapping generation for semantic interoperability. In *Third International Workshop on Database Interoperability (InterDB 2007)*, volume 91.

[Ghebremicael 2017] Ghebremicael, E. S. (2017). Transformation of rest api to graphql for opentosca. Master's thesis, Institute of Architecture of Application Systems - University of Stuttgart, Stuttgart, Germany.

[Gilpin 2015] Gilpin, M. (2015). From the field: The first annual canonical model management forum. forrester blogs. `https://go.forrester.com/blogs/10-03-15-from_the_field_the_first_annual_canonical_model_management_forum`. Acessado em 06/05/2018.

[Gu and Lago 2007] Gu, Q. and Lago, P. (2007). A stakeholder-driven service life cycle model for soa. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pages 1–7. ACM.

[Heuser 2009] Heuser, C. A. (2009). *Projeto de banco de dados: Volume 4 da Série Livros didáticos informática UFRGS*. Bookman Editora.

[Humble and Farley 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education.

[Hüttermann 2012] Hüttermann, M. (2012). *DevOps for Developers*, volume 1. Springer.

[Josuttis 2007] Josuttis, N. M. (2007). *SOA in practice: the art of distributed system design*. "O'Reilly Media, Inc.".

[Marks and Bell 2008] Marks, E. A. and Bell, M. (2008). *Service Oriented Architecture (SOA): a planning and implementation guide for business and technology*. John Wiley & Sons.

[Mehta and Heineman 2002] Mehta, A. and Heineman, G. T. (2002). Evolving legacy system features into fine-grained components. In *Proceedings of the 24th international Conference on Software Engineering*, pages 417–427. ACM.

[Merkel 2014] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[Messina et al. 2016] Messina, A., Rizzo, R., Storniolo, P., and Urso, A. (2016). A simplified database pattern for the microservice architecture. *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*.

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

[Pautasso et al. 2008] Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM.

[Rajkovic et al. 2013] Rajkovic, P., Jankovic, D., and Milenkovic, A. (2013). Using cqrs pattern for improving performances in medical information systems. In *Balkan Conference in Informatics (BCI)*, volume 1036, pages 86–91.

[Richards 2015] Richards, M. (2015). Microservices vs. service-oriented architecture.

[Richardson 2015] Richardson, C. (2015). Introduction to microservices. `https://www.nginx.com/blog/introduction-to-microservices/`. Acessado em 3/5/2018.

[Richardson 2016] Richardson, C. (2016). Microservice architecture patterns and best practices. `http://microservices.io`. Acessado em 3/5/2018.

[Sadalage and Fowler 2012] Sadalage, P. J. and Fowler, M. (2012). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

[Scheibel 2016] Scheibel, G. (2016). A software architecture for storing trajectories using polyglot persistence (in portuguese). *Master dissertation, University of the State of Santa Catarina*.

[Vernon 2013] Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.

[Viennot et al. 2015] Viennot, N., Lécuyer, M., Bell, J., Geambasu, R., and Nieh, J. (2015). Synapse: a microservices architecture for heterogeneous-database web applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 21. ACM.

[Villaca et al. 2018] Villaca, L. H., Azevedo, L. G., and Baião, F. (2018). Query strategies on polyglot persistence in microservices. In *2018 ACM SIGAPP*. ACM.

[Vinoski 2002] Vinoski, S. (2002). Putting the"web"into web services. web services interaction models, part 1. *IEEE Internet Computing*, 6(3):89–91.

[Wampler and Clark 2010] Wampler, D. and Clark, T. (2010). Multiparadigm programming: guest editors' introduction. *IEEE Software*, 27(5):2–7.

[Zimmermann 2016] Zimmermann, O. (2016). Microservices tenets. *Computer Science-Research and Development*, pages 1–10.

## Biografia dos autores

Este trabalho foi escrito pelos seguintes autores.

**Luís H. N. Villaça** é Bacharel em Informática pela UFRJ (1999) e Tecnólogo em Processamento de Dados pelo Centro de Ensino Superior de Juiz de Fora (1994). Cursa o Mestrado na Universidade Federal do Estado do Rio de Janeiro (PPGI/UNIRIO). Trabalha há 20 anos como como analista/desenvolvedor (1996-2006) (CNPQ, Avanti Prima Engenharia, Procwork, Quality Systems, EDS), líder técnico e arquiteto JEE (IBM-2006-2012), auditor interno e arquiteto de SI (Petrobras Transportes-2012-2018). Desde de Agosto de 2017 trabalha em projeto e desenvolvimento de sistemas em arquitetura de Microsserviços, o qual é o tema principal da sua dissertação de mestrado. Detalhes em `http://lattes.cnpq.br/2362414271105479`.

**Antônio Fonseca Pimenta Júnior** é Mestre (2017) em Informática pelo Programa de Pós-Graduação em Informática (PPGI) pela Universidade Federal do Estado do Rio de Janeiro (UNIRIO) e Bacharel em Ciência da Computação (2009) pela Universidade Estadual de Santa Cruz (UESC). Trabalha há 8 anos na área de analise de sistemas como analista/desenvolvedor (CPM Braxis 2010-2013); Analista, Líder tecnico e Gerente de projetos na Dataprev desde 2013. Detalhes em `http://lattes.cnpq.br/1614311113198961`.

**Leonardo G. Azevedo** é pesquisador da IBM Research Brazil desde 2013 e professor da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) desde 2009. Leonardo é Doutor (2005) e Mestre (2001) em Ciências, em Engenharia de Sistemas e Computação pela COPPE(UFRJ) e Bacharel em Informática pela UFRJ (2000). Leonardo tem mais de 20 anos de experiência em pesquisa e desenvolvimento de sistemas tendo atuado em diversas empresas e para o governo. Sua pesquisa concentra-se nos seguintes temas: Sistemas Distribuídos; Arquitetura Orientada a Serviços (SOA); Microsserviços; Gestão de Processos de Negócio (BPM); Computação Cognitiva; e, Bancos de Dados Espaciais. Ele publicou mais de 90 artigos científicos nestes temas em revistas e conferências nacionais e internacionais. Detalhes em `http://researcher.ibm.com/researcher/view.php?person=br-lga` e `http://lattes.cnpq.br/7214791464543522`.