Capítulo 2

Desenvolvimento de Soluções com Serviços: SOA, Cloud e Microsserviços

Leonardo Guerreiro Azevedo

Abstract

Service-Oriented Architecture (SOA), Cloud Computing and Microservices Architecture (MSA) are fundamental concepts, highly interconnected, which should be understood for the development of flexible, distributed, aligned to business and high-performance applications - essential characteristics of modern information systems. The SOA goal is to reduce costs and deadlines by developing applications using existing services composition. In Microservice Architecture, each microservice makes available specific functionalities of a domain. They are developed and deployed independent of each other. In Cloud Computing, processing, storage, data and resources of software are virtualized towards on-demand scaling and availability. This short course has theoretical and practical features. Its goal is to give expertise in distributed application development employing those paradigms for building information systems.

Resumo

Arquitetura Orientada a Serviços (SOA), Computação em Nuvem (Cloud) e Arquitetura de Microsserviços (MSA) são conceitos fundamentais, altamente interligados, que devem ser bem entendidos para o desenvolvimento ágil de aplicações distribuídas flexíveis, alinhadas ao negócio e com desempenho - características essenciais para os sistemas de informação modernos. SOA tem objetivo de reduzir custos e prazos pelo desenvolvimento de aplicações através da composição de serviços. Em MSA, cada microsserviço disponibiliza funcionalidades específicas de um domínio, sendo desenvolvido e implantado independentemente uns dos outros. Na Cloud, processamento, armazenamento, dados e recursos de software são virtualizados em busca de escalonamento e disponibilização sob demanda sendo seu uso um dos princípios de MSA. Este minicurso tem característica teórico-prática e seu objetivo é capacitar no desenvolvimento de aplicações distribuídas empregando estes paradigmas para construção de sistemas de informação.

2.1. Introdução

Arquitetura Orientada a Serviços (SOA - Service-Oriented Architecture) é uma abordagem para construir e integrar aplicações pela descoberta, invocação e composição de serviços distribuídos [Papazoglou and Van Den Heuvel 2007]. SOA permite reduzir custos e prazos pelo desenvolvimento de aplicações através da composição de serviços [Erl 2005]. SOA facilita interoperabilidade entre tecnologias de *middleware* heterogêneas e promove o acoplamento fraco entre consumidor de serviço (requisitantes, clientes) e provedor de serviço [Pautasso et al. 2008]. SOA tem se mostrado como um importante paradigma para desenvolvimento de aplicações flexíveis, distribuídas, alinhadas ao negócio, compostas por serviços independentes de plataforma e protocolo em um ambiente distribuído [Papazoglou and Van Den Heuvel 2007, Erl 2005].

Arquitetura de Microsserviços (MSA - Microservice Architecture) é visto de diferentes formas na literatura (como uma nova abordagem de SOA ou uma evolução de SOA ou uma implementação de SOA) para a utilização de tecnologias para construção de sistemas [Richardson 2016]. Organizações vêm obtendo sucesso na adoção de **MSA**, demonstrando ganhos de eficiência e escalabilidade pela construção de produtos providos por pequenas equipes multidisciplinares, em um ciclo de vida mais curto, e abertos à integração, porém independentes [Fowler and Lewis 2014]. Um desafio de MSA surge a partir da necessidade de integrar dados de origens distintas, o qual 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. Villaça *et al.* apresentam as principais soluções MSA para este problema [Villaça et al. 2018a] e apresenta como fazer este tipo de integração na prática [Villaça et al. 2018b].

Computação na Nuvem permite disponibilizar serviços encapsulando seus detalhes internos. A computação é virtualizada através da construção de componentes distribuídos, tais como, processamento, armazenamento, dados e recursos de software. Nesse ambiente, usuários têm acesso a grande poder de processamento de uma maneira totalmente virtualizada e escalável [Mell et al. 2011].

Este minicurso apresenta os principais conceitos destes três paradigmas de maneira prática. Seu objetivo é proporcionar a fundamentação teórica e prática de SOA, Microsserviços e Cloud. Material complementar está disponível no site deste minicurso [Azevedo 2020] com exemplos de código em diferentes níveis de complexidade, apresentações, exercícios etc.

O principal pré-requisito para o aluno é ter conhecimento em programação orientada a objeto, em especial na linguagem Java. Não é necessário conhecimento avançado na linguagem, uma vez que, quando os conceitos são empregados na parte prática, eles são explicados.

O minicurso está dividido da seguinte forma:

- A Seção 2.2 apresenta a Arquitetura Orientada a Serviços. Esta seção provê o fundamento necessário da área, o qual também é base para entender os conceitos relacionados a microsserviços. É uma seção em maior parte teórica, incluindo motivação, definição e princípios de SOA além de introduzir a tecnologia de Serviços

Web, que é a principal tecnologia para implementação de SOA. A parte prática desta seção refere-se a caracterizar o desenvolvimento de Serviços Web através das tecnologias SOAP e REST. As tecnologias relacionadas são apresentadas e exemplos de implementação são discutidos em detalhes.

- A Seção 2.3 detalha Arquitetura de Microsserviços, a qual é uma forma de implementação e implantação de serviços em SOA empregando práticas do estado da arte de engenharia de software [Zimmermann 2017]. Esta seção apresenta a comparação de Microsserviços com SOA e a comparação de microsserviços com arquitetura monolítica para desenvolvimento de aplicações. Além disso, ela apresenta os princípios que norteiam o paradigma e as principais tecnologias para implantação. Ao final, é apresentado um exemplo de implantação de microsserviço empregando Docker.

- A Seção 2.4 é dedicada à Cloud Computing, a qual é um modelo que permite acesso ubíquo, conveniente, sob demanda a um conjunto compartilhado de recursos de computação (por exemplo, rede, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e liberados com esforço mínimo de gerência ou interação com o provedor do serviço [Mell et al. 2011]. A seção apresenta as características essenciais de Cloud Computing, modelos de implantação e modelos de serviços. Finalmente, a seção apresenta o IBM Cloud, uma ferramenta para desenvolver, implantar, executar e gerenciar aplicações na Cloud.

- A Seção 2.5 apresenta os exercícios práticos a serem seguidos para aprofundamento dos conceitos apresentados nas seções anteriores. A seção aponta para a página do minicurso [Azevedo 2020] onde estão disponibilizados o cenário dos exercícios, passo-a-passo de instalação de ferramentas e de execução dos exercícios, arquivos fontes, apresentações e outros materiais de apoio. O objetivo é que esta página seja evoluída ao longo do tempo de acordo com as aulas lecionadas e o *feedback* dos alunos.

- Finalmente, a Seção 2.6 apresenta as considerações finais.

2.2. Arquitetura Orientada a Serviços

Esta seção está dividida da seguinte forma. Na Section 2.2.1, são apresentados a motivação, definição e princípios de SOA, além das características de serviços. https://overleaf.sl.cloud9.ibm.com/p
A Seção 2.2.2 apresenta a definição de Serviços Web (principal tecnologia para implementação de serviços) e dos seus principais tipos de implementação: Serviços Web SOAP e Serviços Web REST, ilustrando a implementação de cada um destes tipos de serviços.

2.2.1. Definição, princípios e motivação do paradigma

Organizações modernas precisam responder de forma efetiva e rápida às oportunidades do mercado. Suas 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.

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]. Em SOA, funcionalidades do negócio são disponibilizadas para 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]. Um serviço recebe uma mensagem de entrada, a processa e retorna uma mensagem de resposta. Por exemplo, em um sistema bancário teríamos os seguintes serviços: serviço para buscar nomes e endereços; serviço de abertura de conta; serviço de balanço de contas; serviço de depósitos. Dentre as características (ou princípios) de serviços destacamos [Erl 2005]: (i) Deve estar alinhado ao negócio, isto é, fornecer funcionalidade de um processo da organização; (ii) Deve ser operacionalmente independente com alta coesão e acoplamento fraco; (iii) Deve ser sem estado, isto é, fornecer os mesmos resultados para uma mesma entrada; (iv) Deve permitir composição; (v) Deve ser atômico/auto-contido; (vi) Deve garantir consistência das informações; (vii) Deve ter pré e pós-condições bem definidas; (viii) Deve garantir interoperabilidade, ou seja, poder se comunicar com consumidores ou consumir outros serviços desenvolvidos empregando diversificadas tecnologias; (ix) Deve permitir reuso. Estes princípios podem ser relaxados para alcançar determinados objetivos como, por exemplo, serviços compostos não são atômicos, pois utilizam outros serviços para realizar suas tarefas.

Ciclo de vida de uma abordagem tradicional não se aplica diretamente a uma abordagem SOA. SOA herda problemas encontrados no desenvolvimento de software tradicional, como, por exemplo: não satisfação das necessidades dos usuários, uso de recursos além dos orçados, ultrapassagem do tempo estimado para o projeto. Também herda problemas existentes no desenvolvimento de software distribuído, como, por exemplo: complexidade operacional, falta de comunicação entre as pessoas envolvidas, falta de clareza nas tarefas de desenvolvimento, distribuição de responsabilidades. Além disso, SOA traz preocupações adicionais, tais como: cooperação entre os papéis arquiteturais de SOA, bom entendimento dos modelos de negócio e relacionamento entre os parceiros de negócio, como lidar com requisitos conflitantes como distribuir serviços através das fronteiras da organização, como produzir serviços de forma a possibilitar o reuso. Portanto, um ciclo de vida específico para SOA é necessário [Gu and Lago 2007].

Um ciclo de vida de SOA deve considerar os papéis arquiteturais (*stakeholders*): provedor, consumidor e *brokers*/registro de serviços. O *Provedor de Serviços* é o proprietário do serviço. Ele é responsável por desenvolver, publicar e manter serviços para serem consumidos. O *Consumidor de Serviços* é responsável por invocar serviços através de uma aplicação que atende aos requisitos de um usuário final a qual é construída e disponibilizada pelo *Servidor de Aplicações*. É razoável considerar que *Servidor de Aplicações* e *Consumidor de Serviços* atuam de forma combinada, pois o primeiro analisa os requisitos do usuário e projeta, implementa e testa a aplicação que o usuário final irá utilizar. No momento em que serviços são consumidos, este *stakeholder* assume o papel de *Consumidor de Serviços* e descobre serviços, orquestra, negocia, invoca e monitora os mesmos. Já o *Broker de Serviços* provê a informação da localização do serviço que

está contida em um registro de serviços que é mantido por ele. *Provedores de Serviços* utilizam o registro para publicar seus serviços e os *Consumidores de Serviços* para localizá-los. *Broker de Serviços* tem um papel cada vez mais proeminente na redução da lacuna entre os requisitos de negócio e tecnologia [Duan et al. 2014]. Fornecedores de tecnologia proveem diversas ferramentas para registro de serviços, tais como: IBM WebSphere Service Registry and Repository[1], Oracle Enterprise Repository[2], Anypoint service registry (MuleSoft)[3], WSO2 Governance Registry[4].

2.2.2. Serviços Web

Serviço Web é a principal tecnologia para implementação de uma arquitetura orientada a serviços [Josuttis 2007]. Um Serviço Web 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 Serviço Web de acordo com o formato de suas mensagens empregando um protocolo de comunicação (tipicamente HTTP) [Booths et al. 2004]. Existem dois tipos de Serviços Web: Serviço Web SOAP (também conhecidos como Serviço Web WS-* ou "Big" Web Service) e Serviço Web RESTful [Pautasso et al. 2008].

2.2.2.1. Serviços Web SOAP

Serviços Web SOAP empregam um conjunto de tecnologias baseadas em XML[5] (Extensible Markup Language), tais como SOAP, WSDL, XSD e UDDI.

XML 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[6] (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

[1] https://developer.ibm.com/integration/docs/wsrr/

[2] https://www.oracle.com/middleware/technologies/enterprise-repository.html

[3] https://www.mulesoft.com/resources/esb/service-registry-repository

[4] https://wso2.com/products/governance-registry/

[5] https://www.w3.org/TR/xml/

[6] https://www.w3.org/TR/soap12-part1/

framework foi projetado para ser independente de qualquer modelo de programação particular e outas semânticas específicas de implementação.

WSDL[7] (Web Service Description Language) é uma linguagem baseada em XML para especificação do contrato de serviços SOAP. Um documento WSDL de um serviço descreve as operações disponíveis pelo serviço, os esquemas dos tipos de dados referentes aos parâmetros de entrada e de retorno das operações, os *endpoints* onde os serviços estão disponibilizados, o formato de troca de mensagens, o protocolo de comunicação etc. As definições de serviços em WSDL proveem documentação para clientes consumirem os serviços em um ambiente distribuído e serve como uma “receita"para automatizar os detalhes envolvidos na comunicação de aplicações.

A Listagem 2.1 apresenta um exemplo de implementação de Serviço Web SOAP em Java. Nas linhas 3, 4 e 5 temos os *imports* necessários para as anotações *@WebService*, *@WebMethod* e *@WebParam* do pacote JAX-WS (API java para Serviços Web XML). A especificação da API JAX-WS[8] define um padrão para fazer o mapeamento *Java-WSDL*, isto é, a especificação determina como um método Java é invocado e como a mensagem SOAP é mapeada nos parâmetros dos métodos. Este mapeamento também determina como o valor de retorno do método é mapeado em uma mensagem de resposta SOAP. JAX-WS usa anotações para simplificar o desenvolvimento e a implantação de Serviços Web e clientes de Serviços Web. JAX-WS 2.0 substituiu JAX-RPC buscando Serviços Web com mensagens no formato de documento [Hewitt 2009]. *@WebService* é utilizado (Linha 7) para definir a classe *StrManagementSOAPWS* como um serviço disponibilizado como serviço *StrManagementWS*. *@WebMethod* é utilizado (Linha 10) para anotar o método `concat` para ser invocado quando a operação *concatenate* for invocada. *@WebParam* é utilizado (linhas 11 e 12) para definir o nome dos parâmetros do método *str1* and *str2*, respectivamente.

```
1  package org.str.management;
2
3  import javax.jws.WebService;
4  import javax.jws.WebMethod;
5  import javax.jws.WebParam;
6
7  @WebService(serviceName = "StrManagementSOAPWS")
8  public class StrManagementSOAPWS {
9
10   @WebMethod(operationName = "concatenate")
11   public String concat(@WebParam(name = "str1") String str1,
12               @WebParam(name = "str2") String str2) {
13     return str1 + str2;
14   }
15 }
```

Listing 2.1. Exemplo de implementação de Serviço Web SOAP em Java.

Um cliente para o serviço implementado em Java é apresentado na Listagem 2.2. Tanto o serviço como o cliente foram implementados empregando o NetBeans[9], o qual gera o código e as anotações para o serviço e, para o cliente, gera os artefatos java necessários para invocar o serviço.

[7] `https://www.w3.org/TR/wsdl/`
[8] `https://en.wikipedia.org/wiki/Java_API_for_XML_Web_Services`
[9] `https://netbeans.org/`

```
1  package org.str.management;
2
3  public class StrManagementSOAPclient {
4
5    public static void main(String[] args) {
6      String str = concatenate("Curso ", "SOA");
7      System.out.println(str);
8    }
9
10   private static String concatenate(java.lang.String str1, java.lang.String str2) {
11     org.str.management.client.artifacts.StrManagementSOAPWS_Service service = new org
       .str.management.client.artifacts.StrManagementSOAPWS_Service();
12     org.str.management.client.artifacts.StrManagementSOAPWS port = service.
       getStrManagementSOAPWSPort();
13     return port.concatenate(str1, str2);
14   }
15 }
```

Listing 2.2. Exemplo de implementação de cliente em Java para o Serviço Web SOAP.

O código e os slides que explicam as implementações estão disponíveis na página GIT do curso [Azevedo 2020][10]. Neste mesmo local, estão disponíveis outros exemplos de código e aulas para implementação de Serviços Web SOAP, como, por exemplo, implementação de serviços empregando NetBeans e SGBD PostgreSQL[11].

2.2.2.2. Serviço Web REST

Serviço Web REST (Representational State Transfer) é uma alternativa mais simples a Serviços Web SOAP. O estilo REST define um conjunto de restrições/princípios que os serviços devem seguir a fim de alcançar usabilidade, simplicidade, escalabilidade e extensibilidade [Li and Chou 2011]. A ideia central deste estilo é projetar aplicações com baixo acoplamento que se baseiam em recursos, os quais correspondem a qualquer informação que pode ser acessada/manipulada.

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, e tem se estabelecido como o mecanismo para comunicação de sistemas distribuídos [Haupt et al. 2017];

REST[12] 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 interfaces 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 Serviços Web SOAP, mas sim apenas no comportamento dos componentes envolvidos. Dessa forma, o desenvolvedor tem maior flexibilidade em optar pelo formato de mensagem que

[10] pasta *web-services-soap*
[11] `https://www.postgresql.org/`
[12] `https://www.w3.org/TR/ws-arch/\#relwwwrest`

atenda melhor as suas necessidades. Os formatos mais comuns são JSON[13] (Java Script Object Notation), XML e texto puro, mas em teoria qualquer formato pode ser usado. A principal característica de Serviços Web REST é usar explicitamente os métodos HTTP (POST, GET, PUT, DELETE) para denotar a invocação de diferentes operações.

Serviços Web REST se baseiam nos seguintes princípios [Varanasi and Belida 2015] [Fielding 2000]: (i) Separação clara entre as responsabilidades de cliente e servidor a fim de que eles possam evoluir independentemente; (ii) Usar explicitamente os métodos HTTP; (iii) Ser sem estado: chamadas para o serviços devem conter todas as informações necessárias para executá-lo. O servidor não deve armazenar qualquer contexto e dados de sessão devem ser armazenados do lado do cliente; (iv) Expor URIs como uma estrutura de diretório; (v) Respostas do serviço devem ser declaradas como *cacheable* ou não *cacheable*; (vi) Transferir XML, JSON, ou ambos.

A Listagem 2.3 apresenta um exemplo de implementação de Serviço Web REST em Java. Nas linhas 3 a 8, temos os *imports* de JAX-RS (API Java para Serviços Web REST) necessárias à implementação. JAX-RS usa anotações para desenvolvimento de Serviços Web REST as quais permitem o mapeamento de uma classe de recurso (um POJO - Plain-Old Java Object) como um recurso Web[14]. A anotação *@Path* (Linha 10) especifica o caminho relativo de uma classe ou método do recurso. A anotação *@Context* (Linha 13) retorna o contexto inteiro de um objeto, por exemplo, o código *@Context HttpServletRequest request*) retorna o contexto inteiro do objeto *HttpServletRequest*. A anotação *@GET* especifica o método HTTP GET. A anotação *@Produces* (Linha 23) especifica o tipo de mídia de resposta, por exemplo, *text/html*. A anotação *@QueryParam* faz a ligação de um parâmetro do método a um parâmetro (Query) de consulta HTTP.

Para que a uma classe que representa um recurso possa ser disponibilizada como um serviço a configuração apresentada na Listagem 2.4 é necessária. Este código define quais Serviços Web serão disponibilizados na aplicação, o que é feito via a adição da classe do recurso (Linha 17).

```
1  package strmanagement;
2
3  import javax.ws.rs.core.Context;
4  import javax.ws.rs.core.UriInfo;
5  import javax.ws.rs.Produces;
6  import javax.ws.rs.GET;
7  import javax.ws.rs.Path;
8  import javax.ws.rs.QueryParam;
9
10 @Path("strmanagement")
11 public class StrManagementResource {
12
13   @Context
14   private UriInfo context;
15
16   /**
17   * Creates a new instance of StrManagementResource
18   */
19   public StrManagementResource() {
20   }
21
22   @GET
23   @Produces("text/html")
```

[13] https://www.json.org/

[14] https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services

```
24    public String getHtml(
25      @QueryParam("str1") String str1,
26      @QueryParam("str2") String str2){
27        return "<html><body><h1>"+str1+str2+"</body></h1></html>";
28    }
29
30  }
```

Listing 2.3. Exemplo de implementação de Serviço Web REST.

```
1  package strmanagement;
2
3  import java.util.Set;
4  import javax.ws.rs.core.Application;
5
6  @javax.ws.rs.ApplicationPath("webresources")
7  public class ApplicationConfig extends Application {
8
9    @Override
10   public Set<Class<?>> getClasses() {
11     Set<Class<?>> resources = new java.util.HashSet<>();
12     addRestResourceClasses(resources);
13     return resources;
14   }
15
16   private void addRestResourceClasses(Set<Class<?>> resources) {
17     resources.add(strmanagement.StrManagementResource.class);
18   }
19 }
```

Listing 2.4. Configuração necessária para disponibilização Serviço Web REST.

Um cliente para o serviço REST implementado em Java é apresenta na Listagem 2.5 e na Listagem 2.6. A primeira listagem corresponde ao código necessário para configurar o acesso Serviço Web alvo (linhas 8 a 24) e para fazer a invocação (Linha 25). A segunda listagem, basicamente, é um exemplo de uso da implementação anterior através de um console Java.

```
1  package concatclientapplication;
2
3  import javax.ws.rs.ClientErrorException;
4  import javax.ws.rs.client.Client;
5  import javax.ws.rs.client.WebTarget;
6
7  public class StrManagementResourceClient {
8    private final WebTarget webTarget;
9    private final Client client;
10   private static final String BASE_URI = "http://localhost:8080/
       StrManagementRESTWSApp/webresources";
11
12   public StrManagementResourceClient() {
13     client = javax.ws.rs.client.ClientBuilder.newClient();
14     webTarget = client.target(BASE_URI).path("strmanagement");
15   }
16
17   public String getHtml(String str1, String str2) throws ClientErrorException {
18     WebTarget resource = webTarget;
19     if (str1 != null) {
20       resource = resource.queryParam("str1", str1);
21     }
22     if (str2 != null) {
23       resource = resource.queryParam("str2", str2);
24     }
25     return resource.request(javax.ws.rs.core.MediaType.TEXT_HTML).get(String.class);
26   }
27 }
```

Listing 2.5. Exemplo de implementação de cliente para o Serviço Web REST.

```
1  package concatclientapplication;
2
3  public class ConcatClientApplication {
4
5    public static void main(String[] args) {
6      StrManagementResourceClient strManagement=new StrManagementResourceClient();
7
8      String str = strManagement.getHtml("Barmenia", "Versicherungen");
9
10     System.out.println("concat: " + str);
11   }
12 }
```

Listing 2.6. Exemplo de implementação de invocação do Serviço Web REST.

Existem diferentes tecnologias para especificação de contratos de serviços REST, tais como: Swagger (ou OpenAPI)[15], WADL[16], WSDL2.0[17], e API Blueprint[18]. A principal delas é Swagger, cuja terceira versão foi renomeada para especificação OpenAPI (OAS - OpenAPI Specification) [OpenAPI initiative 2018]. OpenAPI é a linguagem mais popular para especificação de contratos REST [Tsouroplis et al. 2015] e a iniciativa OpenAPI provê um *framework* completo para criação da documentação e geração de código do cliente e do servidor a partir da especificação. Existem várias ferramentas para especificação de contratos REST em Swagger/OpenAPI. Santos *et al.* apresentam uma análise das ferramentas cujos resultados ajudar engenheiros de software a escolher as ferramentas mais adequadas e apresentam lacunas a serem tratadas em iniciativas de pesquisa [Santos et al. 2020].

Os códigos fontes e os slides que explicam os conceitos em detalhes e implementações de serviços REST estão disponíveis na página curso [Azevedo 2020][19].

2.3. Arquitetura de Microsservicos

Esta seção apresenta os principais conceitos de microsserviços, tais como: definição e como microsserviços e SOA estão relacionados (Seção 2.3.1); princípios (Seção 2.3.2); comparação entre arquitetura monolítica e arquitetura de microsserviços (Seção 2.3.3); algumas principais tecnologias para implantação de microsserviços (Seção 2.3.4); e, um exemplo de implantação de um Serviço Web em contêiner Docker (Seção 2.3.5).

2.3.1. Definição

A Arquitetura de Microsserviços (MSA), 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]. É uma proposta de construção de pequenas aplicações, desenvolvidas

[15] http://swagger.io
[16] https://www.w3.org/Submission/wadl/
[17] https://www.w3.org/TR/2007/REC-wsdl20-20070626/
[18] http://raml.org
[19] Pasta *web-services-rest*

de forma independente, que tendem a apresentar aspectos de processamento e de interoperabilidade eficientes, permite a implantação/entrega contínua de aplicações grandes e complexas. Ela permite que uma organização evolua em sua *stack* de tecnologia de maneira mais fácil.

Apoiadores de microsserviços afirmam que este é um novo estilo arquitetural [Richards 2015]; em contraste, aqueles que defendem SOA argumentam que microsserviços é uma abordagem de implementação e implantação de SOA [Zimmermann 2017]. Neste trabalho, seguimos a definição de que microsserviços é uma forma de implementação e implantação de serviços em SOA empregando práticas do estado da arte de engenharia de software de acordo com revisão feita na literatura [Zimmermann 2017]. Esta revisão concluiu que as diferenças entre microsserviços e tentativas anteriores de aplicar SOA não estão relacionadas ao estilo arquitetural (isto é, restrições e intenções de projeto e princípios de independência de plataforma e padrões), mas sim na realização da arquitetura (por exemplo, paradigmas de desenvolvimento e implantação e tecnologias).

O termo microsserviços apareceu inicialmente em blogs e artigos online a partir de 2014; Lewis e Fowler [Fowler and Lewis 2014] apresentam uma coleção de postagens sobre o tema. A partir de 2016 que microsserviços começaram a aparecer em workshops e conferências [Zimmermann 2017].

Arquitetura de Microsserviços é uma abordagem para o desenvolvimento de uma única aplicação formada por um conjunto de pequenos serviços (microsserviços), cada um executando em seu próprio processo (por exemplo, isolado em um 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 (projetado usando técnicas de projeto orientadas ao domínio ou Domain-Driven Design -DDD) e são implantados de forma completamente automatizada (utilizando técnicas de automação e conteinerização). Eles podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias de banco de dados (isto é, múltiplos paradigmas e tecnologia que melhor se aplica como solução). Existe uma camada mínima centralizada de gerenciamento desses serviços [Fowler and Lewis 2014].

O estilo de disponibilização de funcionalidades como microsserviços é análogo aos utilitários UNIX, os quais são desenvolvidos cada um com uma finalidade específica e podem ser combinados para executar tarefas complexas. Por exemplo, combinar os utilitários *grep*, *cat* e *find* através de um *script shell* [Richardson 2016].

2.3.2. Princípios

Há sete princípios atribuídos a microsserviços [Zimmermann 2017, Fowler and Lewis 2014, Newman 2015]:

- Desenvolver microsserviços como unidades com **interfaces de granularidade fina** com responsabilidade única que encapsulam dados e lógica de processamento e são expostos remotamente, tipicamente através de APIs Web (por exemplo, recursos HTTP RESTful) ou filas de mensagens assíncronas. Estas unidades podem ser implantadas, alteradas, substituídas e escalonadas independentemente uma das outras.
- Usar **práticas de desenvolvimento e linguagens padronizadas orientadas ao ne-**

gócio paraidentificar e conceitualizar 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;

- Seguir **princípios de projeto baseados em computação em nuvem**, por exemplo, *IDEAL* (isolamento, distribuição, elasticidade, gestão automatizada e baixo acoplamento)[Fehling et al. 2014].

- Empregar **múltiplos paradigmas** de computação e armazenamento (isto é, usar estratégia de programação e persistência poliglota) [Wampler and Clark 2010] que melhor se apliquem ao desenvolvimento da solução.

- Usar **contêineres leves** para disponibilizar serviços para promover artefatos através de processos de entrega contínua (*e.g.*, Docker [Merkel 2014]).

- Empregar técnicas de **entrega contínua descentralizada** para promover um alto nível de automação e autonomia. Isto demanda maturidade para construir artefatos de forma automatizada, aliados a uma suíte de testes [Humble and Farley 2010].

- Empregar ***DevOps*** através do uso de técnicas e automatização de configuração, desempenho e gerenciamento de falhas, o qual estende práticas ágeis e incluem monitoramento de serviços [Hüttermann 2012].

2.3.3. Arquitetura Monolítica x Arquitetura de Microsserviços

Microsserviços são uma alternativa às aplicações monolíticas onde uma única aplicação contém toda a lógica do negócio e gerenciamento de dados [Richardson 2015]. Neste caso, mesmo a aplicação tendo uma arquitetura interna dividida em módulos, ela é empacotada e disponibilizada inteiramente, por exemplo, um WAR[20] 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 e distribuídas. Inclusive elas podem ser escalonadas colocando-se múltiplas cópias em múltiplos servidores e empregando balanceamento de carga.

Problemas começam a surgir quando a aplicação "monolítica" cresce tornando-se extremamente complexa. Corrigir erros (*bugs*) e implementar novas funcionalidades corretamente tornam-se difíceis e consomem muito tempo. A carga cognitiva para entender o código é grande. Quando à implantação, a aplicação monolítica grande pode demorar a iniciar e alterações no código ao longo do dia, demandando reimplantação (que deve ser inteira) pode demandar muito tempo para a aplicação e suas cópias (quando há escalonamento) estarem ativas [Richardson 2015].

[20]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.

Em aplicações monolíticas, seus módulos executam no mesmo processo. Logo, erro em um módulo pode derrubar toda a aplicação. A adoção de novos *frameworks* também torna-se um problema, por não poder facilmente ser executada em uma parte específica do código o qual não está inteiramente dependente.

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 expondo/-consumindo funcionalidades para/de outros microsserviços. Alguns microsserviços podem implementar interfaces Web permitindo implantar experiências distintas (independentes) para usuários ou dispositivos específicos ou para casos de uso específicos.

A fim de reduzir o acoplamento, em geral, cada microsserviço tem seu próprio banco de dados. Eventualmente, dados podem ter que ser replicados e/ou 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].

A arquitetura de microsserviços trata o problema da complexidade através da decomposição da aplicação monolítica em um conjunto de microsserviços. Ela inverte a lei de Conway que afirma que organizações que projetam sistemas ficam limitadas a produzir projetos que são cópias da estrutura de comunicação da organização [Conway 1968]. Em microsserviços, times são organizados de forma a serem responsáveis por um único microsserviços desde o desenvolvimento até a implantação. Portanto, isto mitiga os problemas de comunicação de organizações trabalhando em bases de código grandes identificada na lei de Brook do mítico homem mês [Brooks Jr 1995].

Existem várias vantagens da abordagem de microsserviços. O escopo menor de cada microsserviço minimiza a carga cognitiva dos desenvolvedores porque o código base de trabalho é pequeno e fácil de entender. Código base menor facilita antecipar o impacto de mudanças. A equipe também tem liberdade de escolher a tecnologia que melhor resolve a tarefa; enquanto que, em um código base grande de uma aplicação monolítica, os desenvolvedores de um módulo podem escolher as melhores ferramentas desde que estejam de acordo com a linguagem e os *frameworks* do monólito. Como microsserviço foca em uma funcionalidade específica, fica mais fácil prever o seu comportamento quanto às características de execução, tais como, CPU e escrita em disco (*I/O*), quanto ao escalonamento de recursos de acordo com as requisições, quanto à sensibilidade à latência, se ele é sem estado e como ele é favorável a balanceamento de carga. Portanto, operadores tem mais informações para tomar decisões de como alocar recursos baseados nos características de cada microsserviço [Azevedo et al. 2019].

Existem também desvantagens para microsserviços. Do ponto de vista do desenvolvedor, perde-se a capacidade de usar a depurador da IDE[21] para observar todas as interações com outras partes do sistema. Microsserviços requer o uso pesado de ferramentas para *logging*, rastreamento e monitoramento de desempenho para dar a visibilidade do que está acontecendo. Do ponto de vista do operador, a gerência de configuração re-

[21] Integrated Development Environment

quer executar e atualizar dezenas ou centenas de de microsserviços colaborando o que é desafiante. Plataformas de orquestração de contêineres, *middlewares* e ferramentas de integração contínua e metodologias como DevOps ajudam a sobrecarga dos operados, mas trazem complexidade que deve ser tratada é ainda muito maior do que em arquiteturas monolíticas [Azevedo et al. 2019].

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 inter processo, o que é mais complexo do que invocar métodos no nível da linguagem, sendo também mais complexo de testar, por exemplo, é necessário implementar testes de integração entre os microsserviços. Particionamento do banco de dados entre os microsserviços e persistência poliglota traz o problema de garantia de integridade dos dados distribuídos, entre outros desafios. Newman apresenta soluções para integração de dados com bancos de dados poliglotas [Newman 2015]. Villaça *et al.* analisam as soluções indicadas por Newman e indica o uso de modelo de dados canônico como uma possível solução para este problema. Eles se baseiam em padrões definidos antes do termo microsserviços ser definido, mas cujas características tem muita relação com este paradigma [Villaça et al. 2018a]. Villaça *et al.* ilustram o uso destes padrões [Villaça et al. 2018b] e Villaça *et al.* apresentam uma proposta de integração de dados provenientes de fontes de dados heterogêneas empregando modelo dados canônico [Villaça et al. 2020]. Além disso, Azevedo *et al.* [Azevedo et al. 2019] apresentam um exemplo de como fazer a integração de dados empregando o modelo CQRS (Command-Query Responsability Segregation) [Fowler 2011] em um cenário real na área de Óleo & Gás.

2.3.4. Tecnologias para implantação de microsserviços

Microsserviços são desenvolvidos como Serviços Web (Seção 2.2.2), principalmente, Serviços Web RESTful como ressaltado em no princípio "interfaces de granularidade fina" (Seção 2.3.2). Dentre as tecnologias mais utilizadas para implantação e orquestração de microsserviços temos, por exemplo, Docker, Docker Swarm, Kubernetes, Apache Mesos e ZooKeeper, que são tecnologias que seguem o princípio "contêineres leves". Ferramentas para Integração contínua (*continuous integration* - CI) e entrega contínua (*continuous deployment* - CD) são amplamente utilizadas, tais como, Jenkins, Hudson e Chef. Integração contínua envolve automatizar a integração de código em um repositório compartilhado uma vez ao dia ou mais e entrega contínua refere-se a automatizar a implantação de pacotes do software no ambiente de implantação - princípios "entrega contínua descentralizada'" e "DevOps". Ferramentas para monitoramento de serviços incluem, por exemplo, as ferramentas Elastic[22] - elasticsearch, logstack e kibana.

[22] `https://github.com/elastic/`

2.3.4.1. 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 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*.

Dentre as vantagens de contêineres temos: (i) flexíveis: permitem empacotar diversos tipos de aplicações; (ii) leves: compartilham o *kernel* da máquina hospedeira sendo mais rápidos do que máquinas virtuais; (iii) portáveis: são construídos localmente, mas podem ser implantados na Cloud e executar em qualquer lugar; (iv) têm baixo acoplamento: são alto suficientes e encapsulados, permitem serem substituídos e atualizados em impactar outros contêineres; (v) escaláveis: replicas de contêineres podem ser aumentadas e distribuídas automaticamente; (vi) seguros: possuem restrições e isolamento.

2.3.4.2. Docker

Docker[23] 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. Uma imagem Docker provê tudo o que é necessário para executar uma aplicação, por exemplo, código ou binário, *runtimes*, dependências e objetos de sistemas de arquivo necessários.

Na página do Docker, é apresentado um tutorial[24] para instalar o ambiente, construir uma imagem e executá-la como um contêiner, configurar e usar as ferramentas de orquestração Kubernetes e Swarm e compartilhar aplicações conteinerizadas no Docker Hub. Nesta seção, apresentamos algumas características que ajudam a realizar este tutorial.

Em geral, o fluxo de desenvolvimento de uma aplicação conteinerizada inclui os seguintes passos: (i) Criar e testar contêineres individualmente para cada componente de

[23] `https://www.docker.com/`
[24] `https://docs.docker.com/get-started/`

sua aplicação iniciando pela criação de imagens Docker; (ii) Combinar seus contêineres e infraestrutura de apoio em uma aplicação completa, expressa como um arquivo Docker (i.e., *docker stack file*) ou um arquivo YAML Kubernetes; (iii) Testar, compartilhar e implantar sua aplicação conteinerizada completa.

O Dockerfile é um arquivo que descreve o passo-a-passo para se construir 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 Listagem 2.7[25] apresenta um exemplo de Dockerfile. O comando da linha 1 indica que a imagem *node:6.11.5* é a imagem base. Esta imagem é uma imagem oficial, construída pelos fornecedores do noje.js e validade pelo Docker, contendo o interpretador node 6.11.5 e dependências básicas. O comando da linha 3 usa *WORKDIR* para indicar que todas as ações subsequentes serão feitas no diretório */usr/src/app* do sistema de arquivos da imagem (e nunca na máquina hospedeira). O comando da linha 4 usa o comando *COPY* para copiar o arquivo *package.json* para a raiz, isto é, */usr/src/app*. O comando da linha 5 executa o comando *npm install* dentro do sistema de arquivos da imagem, o qual lerá o arquivo *package.json* para identificar as dependências do código fonte da aplicação e instalá-las no sistema de arquivo da imagem. Na linha 6, o comando copia todo o restante do conteúdo do diretório corrente da máquina hospedeira para o sistema de arquivos da imagem. Finalmente, na linha 7, usando *CMD* são passados metadados para a imagem descrevendo como executar um contêiner baseado nesta imagem. Estes metadados informam que o processo conteinerizado que esta imagem apoia é *npm start*, ou seja, indica ao Docker para executar *npm start* quando o contêiner inicia. Existem várias outras diretivas[26] para construção de imagens Docker.

```
1  FROM node:6.11.5
2
3  WORKDIR /usr/src/app
4  COPY package.json .
5  RUN npm install
6  COPY . .
7
8  CMD [ "npm", "start" ]
```

Listing 2.7. Exemplo de Docker file.

Usando o comando *build*, a imagem é construída a partir do *Dockerfile* e o comando *run* inicia o container a partir da imagem e comando *rm* apaga o contêiner. (Listagem 2.8[27]). O parâmetro *publish* indica ao Docker para encaminhar para o contêiner na porta 8080 todo o tráfego que chega na porta 8000. O parâmetro *detach* indica que o contêiner deve executar em *background*. O parâmetro *name* define o nome *bb* como o nome do contêiner para ser usado em comandos futuros.

```
1  docker image build -t bulletinboard:1.0 .
2
3  docker container run --publish 8000:8080 --detach --name bb bulletinboard:1.0
4
```

[25] https://docs.docker.com/get-started/part2/
[26] https://docs.docker.com/engine/reference/builder/
[27] https://docs.docker.com/get-started/part2/

```
5 docker container rm --force bb
```

Listing 2.8. Exemplos de comandos para criar imagem docker e iniciar contêiner.

Outra ferramenta importante de Docker é o Docker Hub[28] o qual 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.

2.3.4.3. Kubernetes

Kubernetes proveem várias ferramentas para escalonar, colocar na rede, tornar seguro e manter uma aplicação conteinerizada. Todos os contêineres em Kubernetes são programados como *pods* que são grupos de contêineres co-localizados que compartilham alguns recursos. Objetos Kubernetes são descritos em manifestos chamados de arquivos *Kubernetes YAML*. Estes arquivos descrevem todos os componentes e configurações de uma aplicação Kubernetes e são usados para criar e destruir a aplicação em qualquer ambiente Kubernetes. Aplicações são programadas como *deployments*, os quais, em geral, são grupos de *pods* mantidos automaticamente pelo Kubernetes.

Um arquivo Kubernetes YAML, como ilustrado na Listagem 2.9[29], em geral, contém: *apiVersion*, que indica a API Kubernetes que faz o *parser* do objeto; *kind*, que indica o tipo de objeto, por exemplo, *Deployment* ou *Service*; *metadata*, como, por exemplo, nome de objetos; *spec*, que especifica os parâmetros e configurações do objeto.. No exemplo, o objeto *Deployment* define uma única réplica do pod, o qual é descrito a partir da chave *template*, em apenas um contêiner basedo na imagem *bulletinboard:1.0*. O serviço *NodePort* faz o roteamento do tráfego da porta 30001 da máquina hospedeira para a porta 8080 dentro dos pods permitindo alcançar a aplicação *bulletin board* a partir da rede.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: bb-demo
5    namespace: default
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10       bb: web
11     template:
12       metadata:
13         labels:
14           bb: web
15       spec:
16         containers:
17         - name: bb-site
18            image: bulletinboard:1.0
19 ---
20 apiVersion: v1
21 kind: Service
22 metadata:
23   name: bb-entrypoint
```

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

[29] https://docs.docker.com/get-started/part3/

```
24    namespace: default
25  spec:
26    type: NodePort
27    selector:
28      bb: web
29    ports:
30    - port: 8080
31      targetPort: 8080
32      nodePort: 30001
```

Listing 2.9. Exemplos arquivo Kubernetes YAML.

A Listagem 2.10 apresenta comandos para: fazer a implantação da aplicação (*apply*); verificar se o pod está executando (*get deployments*); verificar se o serviço está executando (*get services*). Caso esteja executando, teste o serviço acessando *localhost:30001* no Browser; destruir a aplicação (*delete*).

```
1  kubectl apply -f bb.yaml
2
3  kubectl get deployments
4
5  kubectl get services
6
7  kubectl delete -f bb.yaml
```

Listing 2.10. Exemplos de comandos para implantação da aplicação com Kubernetes.

2.3.5. Implantando Serviços com Docker

A Listagem 2.11 e a Listagem 2.12 apresentam exemplo de código para disponibilizar os pacotes dos serviços (isto é, os arquivos *WAR* gerados quando do deploy dos serviços) em uma imagem Docker. Os arquivos *Dockerfile*, *start.sh* e *WAR* estão disponíveis na página do curso [Azevedo 2020] pasta *microservices* de exemplos de empacotamento de serviços via contêiner.

```
1
2  FROM glassfish:latest
3
4  COPY StrManagementRESTWSApp.war /
5  COPY start.sh /
6
7  EXPOSE 8080
8
9  ENTRYPOINT ["/start.sh"]
```

Listing 2.11. Implantação do serviço REST usando Docker.

```
1
2  #!/bin/sh
3
4  /usr/local/glassfish-4.1/bin/asadmin start-domain
5  /usr/local/glassfish-4.1/bin/asadmin -u admin deploy /aot.war
6  /usr/local/glassfish-4.1/bin/asadmin stop-domain
7  /usr/local/glassfish-4.1/bin/asadmin start-domain --verbose
```

Listing 2.12. Arquivo para iniciar o servidor.

```
1
2  docker image build -t glassfish:latest .
3
4  docker container run --publish 8000:8080 --detach --name wsrest glassfish:latest
```

Listing 2.13. Comandos para criar a imagem e iniciar o contêiner.

2.4. Cloud Computing

Esta seção apresenta a definição de Cloud Computing e suas características essenciais (Seção 2.4.1), os modelos de implantação (Seção 2.4.2), modelos de serviço (Seção 2.4.3) e um exemplo de plataforma de Cloud Computing (o IBM Cloud - Seção 2.4.4.)

2.4.1. Definição

Cloud Computing é um modelo para permitir acesso ubíquo, conveniente, sob demanda a um conjunto compartilhado de recursos de computação (por exemplo, rede, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e liberados com esforço mínimo de gerência ou interação com o provedor do serviço [Mell et al. 2011].

A característicasessenciais deste modelo são [Mell et al. 2011]:

- **Autoatendimento sob demanda**: As capacidades de computação (por exemplo, tempo de servidor e armazenamento) são provisionados automaticamente sem interação humana com o provedor do serviço.
- **Acesso amplo à rede**: Recursos são disponibilizados na rede e acessados através de mecanismos padronizados via plataforma cliente leve (por exemplo, telefones móveis, *tablet*, *laptops*).
- **Agrupamento de recursos**: Recursos são agrupados para servirem múltiplos consumidores usando um modelo multi-cliente. Eles são dinamicamente atribuídos de acordo com a demanda do consumidor.
- **Elasticidade rápida**: Capacidades podem ser provisionadas e liberadas elasticamente.
- **Serviço de medida**: O uso dos recursos é monitorado, controlado e comunicado, provendo transparência para ambos consumidor e provedor.

2.4.2. Modelos de Implantação da Cloud

Os modelos de implantação da Cloud são:

- **Cloud privada**: Cloud de uso exclusivo por uma única organização envolvendo múltiplos consumidores (por exemplo, unidades de negócio). A Cloud é de propriedade, gerenciada e operada pela própria organização ou por terceiro ou pela combinação de ambos. Ela pode estar hospedada na organização ou fora dela.
- **Cloud comunitária**: Cloud de uso exclusivo de uma comunidade composta por organizações com necessidades e preocupações compartilhadas (por exemplo, políticas de segurança). Cloud de propriedade, gerenciada e operada pela própria comunidade ou por terceiro ou pela combinação de ambos. Ela pode estar hospedada na organização ou fora dela
- **Cloud pública**: Cloud de uso aberto para o público em geral. Ela é de propriedade, gerenciada e operada por uma organização de negócio, acadêmica ou governamental ou pela combinação dos mesmos. Ela existe hospedada no provedor da Cloud.

- **Cloud híbrida**: Composição de duas ou mais infraestruturas distintas da Cloud que permanecem como entidades únicas. Infraestruturas são combinadas por tecnologia padronizada ou proprietária.

2.4.3. Modelos de Serviço da Cloud

Os principais modelos de serviço disponíveis em Cloud Computing são IaaS (Infrastructure as a Service), PaaS (Platform as a Service) e SaaS (Software as a Service) [Mell et al. 2011]:

- **Infrastructure-as-a-Service (IaaS)**: O provedor fornece recursos de computação fundamentais como, por exemplo, processamento, armazenamento e rede. Os consumidores ficam aptos a implantar e executar software arbitrário, tais como sistemas operacionais e aplicações. Exemplos de provedores IaaS são: IBM Cloud[30], Amazon Web Services[31] (AWS), Microsoft Azure[32], Google Cloud[33] OpenNebula[34].

- **Platform-as-a-Service (PaaS)**: O provedor fornece recursos a consumidores para criar ou implantar aplicações criadas usando estes recursos como, por exemplo, linguagens de programação, bibliotecas, serviços e ferramentas de implantação. Recursos compatíveis provenientes de outras fornecedores também podem ser usados. Exemplos de PaaS são : IBM Cloud, Google Cloud, SalesForce.com[35].

- **Software-as-a-Service (SaaS)**: O provedor fornece capacidades para consumidores usarem aplicações executando na infraestrutura da Cloud. Estas aplicações são acessíveis de vários dispositivos clientes usando uma camada de cliente leve (por exemplo, web browser ou API). Exemplos de SaaS são: Google docs[36], Office 365[37], Gmail[38].

O tempo é crítico para as aplicações de hoje em dia as quais devem ter velocidade e agilidade para atender às necessidades dos seus funcionários e/ou clientes e estarem a frente de seus competidores. A Tabela 2.1 ilustra os modelos de serviço em relação a o que é gerenciado pelo cliente e o que é gerenciado pelo provedor da cloud.

O núcleo de TI (ou ***core IT***) de uma organização representa tudo o que ela possui e gerencia em suas centrais de dados (*data centers*). Isto é ainda uma parte crítica da organização. Ela tem **benefícios**, tais como: (i) permite se ter uma versão estável e customizável de acordo com as necessidades do cliente, apesar de ser limitada aos custos; (ii) é necessária para alguns casos (por exemplo, processamento de certas transações); (iii) considera muito dos investimentos já feito pelas organizações (por exemplo, sistemas

[30] https://www.ibm.com/cloud
[31] https://aws.amazon.com/
[32] https://azure.microsoft.com/\textbf{text}
[33] https://cloud.google.com/
[34] https://opennebula.org/
[35] https://www.salesforce.com/
[36] https://docs.google.com/
[37] https://products.office.com/
[38] https://www.gmail.com/

Tabela 2.1. Recursos gerenciados pelo cliente (em fundo branco) e recursos gerenciados pelo provedor da Cloud (em fundo cinza) de acordo com os modelos de serviço da Cloud.

TI principal)	IaaS	PaaS	SaaS
Código	Código	Código	Código
Dados	Dados	Dados	Dados
Runtime	*Runtime*	*Runtime*	*Runtime*
Middleware	*Middleware*	*Middleware*	*Middleware*
S.O.	S.O	S.O.	S.O.
Virtualização	Virtualização	Virtualização	Virtualização
Servidores	Servidores	Servidores	Servidores
Armazenamento	Armazenamento	Armazenamento	Armazenamento
Rede	Rede	Rede	Rede

e dados). Por outro lado, há um grande **comprometimento de tempo** neste caso e como **limitações** temos: (i) tipicamente leva semanas para configurar e implantar uma aplicação inicial; (ii) a organização deve manter e atualizar hardware e software, por exemplo, *packs* de serviços, antivírus e *patches*; (iii) custo pode ser alto; (iv) é necessário equipe dedicada; (v) há dificuldades em experimentar novas tecnologias.

IaaS permite implantação rápida, configuração rápida do ambiente etc. ao abstrair a infraestrutura das tarefas do cliente. Como **benefícios** de IaaS temos: (i) rede, armazenamento, servidores e virtualização são gerenciados pelo provedor de serviços; (ii) oferta da Cloud é mais customizável; (iii) são soluções onde é necessário customização de sistema operacional, *middleware* e *runtime*. Por outro lado, o **comprometimento de tempo** é menor, pois leva-se minutos para prover uma nova máquina virtual e como **limitações** temos: (i) cliente configura e gerencia sistema operacional, *middleware* e *runtime* o que pode demandar dias de trabalho; (ii) manutenções e atualizações também são necessários, tais como, *packs* de serviços, antivírus e *patches*.

Existem casos em que clientes necessitam maior agilidade do que as possibilidades anteriores e não precisam gastar tempo gerenciando plataforma (por exemplo, máquinas virtuais, sistemas operacionais e *runtime*). Neste caso, **PaaS** é uma solução mais adequada. Ela traz como **benefícios**: (i) rapidez na configuração de ambientes e implantação de aplicações; (ii) o provedor do serviços gerencia infraestrutura e plataforma. Quanto ao **comprometimento de tempo**, leva-se minutos para configurar e implantar aplicações. O cliente foca nas aplicações e nos seus dados, ou seja, construir e manter aplicações.

Finalmente, em uma **SaaS**, encontramos como benefícios: (i) infraestrutura, plataforma e aplicações são gerenciados pelo provedor de serviços, o que inclui, por exemplo, atualizações de software, consertos de hardware e ajustes da rede; (ii) a aplicação está totalmente da Cloud e podem ser acessadas de diferentes *endpoints*, dependendo do provedor de serviços. Quanto ao **comprometimento de tempo**, o foco do cliente está no **uso** das aplicações.

2.4.4. IBM Cloud

IBM Cloud[39] corresponde a uma plataforma para desenvolver, implantar, executar e gerenciar aplicações. Ela combina PaaS e IaaS. Ela possui mais de 190 serviços para serem utilizados para construir aplicações. Contém um conjunto de dados e ferramentas avançadas de IA. A plataforma é capaz de apoiar tanto times pequenos de desenvolvimento e organizações de pequeno porte como negócios de grandes empresas.

Dentre os principais componentes da plataforma, temos:

- Console: Permite criar, visualizar e gerenciar recursos da cloud;
- Gerente de identidade e acesso: Permite autenticação segura de usuários;
- Catálogo de aplicações;
- Mecanismos de busca e rotulação: Facilitam filtrar e identificar recursos;
- Camada de disponibilização (*provisioning*): Permite controlar e gerenciar recursos;
- Cobrança: Permite gestão de conta unificada, precificação, medição e relatórios de uso.

O IBM Cloud oferece várias opções para servidores de aplicação, tais como:

- Servidores *Bare Metal*: servidores dedicados de apenas um *tenant* onde nada é compartilhado, incluindo recursos dos servidores, com outros clientes;
- Servidores virtuais: Servidores escaláveis que são comprados com allocação de memória e *cores*.
- Soluções VMware: Permite integrar ou migrar cargas de trabalho VMware da empresa (*on-premise* usando infraestrutura escalável, segura e de alto desempenho e tecnologia de virtualização híbrida da VMware;
- Serviços Kubernetes: Combina contêineres Docke, tecnologia Kubernetes e segurança e isolamento integrados para automatizar implantação, operação, escalonamento e monitoramento de aplicações conteinerizadas em um cluster de computação.
- IBM Cloud Foundry: Permite instanciar múltiplas plataformas Cloud Foundry sob demanda de forma isolada.
- Funções Cloud: Plataforma Function-as-a-Service (FaaS) baseado em Apache OpenWhisk.

Como opções de armazenamento, a plataforma oferece:

- Block Storage: Armazenamento iSCI com persistência em alto desempenho provida e gerenciada independente de instâncias de computação.

[39] `https://www.ibm.com/cloud`

- File Storage: Persistência rápida e flexível atrelada à rede baseada em NFS.

- IBM Cloud Object Storage: Armazenamento de informações usando tecnologia IBM Cloud Object Storage. O armazenamento é encriptado e espalhado em múltiplas localizações geográficas e acessado via API REST.

- IBM Cloud Master Data Management: Armazenamento *offload* de grandes quantidades de dados das centrais de dados da empresa para Cloud Object Storage.

- IBM Cloud Backup: Ferramenta para fazer backup de dados entre servidores localizados na rede da IBM Cloud. O agente de backup é automatizado e gerenciado através de uma ferramenta no Browser.

Para uso da plataforma, pode-se criar uma conta[40] "lite" com vários serviços livres que podem ser usados para experimentar o ambiente. Para serviços mais avançados é necessário um plano pago. No entanto, existe ferramenta para estimar custos[41]. A plataforma oferece *stencils*[42] (i.e., *templates*) para criar diagramas da arquitetura criada usando ferramentas populares de diagramação.

Após o usuário logar na plataforma[43], ele entra no *Dashboard* onde ele pode criar recursos, criar aplicações, adicionar usuários e aprender [44] sobre a plataforma. Uma boa forma de iniciar é através dos tutoriais disponíveis[45], por exemplo o tutorial "Aplicações Web Escaláveis no Kubernetes"[46] que engloba os conhecimentos apresentados neste tutorial ou o tutorial "Entrega Contínua para o Kubernetes"[47]. Também pode-se iniciar o aprendizado a partir de uma linguagem de programação como, por exemplo, Java[48].

No catálogo de aplicação existem várias categorias classificadas como serviços e/ou software, tais como:

- Inteligência artificial: (i) Tradutor de idiomas (Language Translator): tradutor de texto, documentos e páginas Web; (ii) Ideias para literatura médica (Insights for Medical Literature): pesquisa em um corpus da literatura médica e descobre conhecimento/ideias; (iii) Catálogo de Conhecimento (Knowledge Catalog): permite realizar descobertas, catalogar e compartilhar de forma segura os dados da empresa e auxilia em ciência de dados e conformidade de dados; (iv) Watson Studio: provê um conjunto de ferramentas e um ambiente colaborativo para cientistas de dados, desenvolvedores e especialistas de domínio. Pode-se utilizar ferramentas de código aberto como RStudio ou Jupyter Notebooks e serviços do Watson para criar modelos flexíveis. Trata as fases de preparação de dados, preparação dos dados

[40] `https://cloud.ibm.com/registration`
[41] https://github.com/ibm-cloud-architecture/infrastructure-design-decision-tool/
[42] `https://github.com/ibm-cloud-architecture/ibm-cloud-stencils`
[43] `https://cloud.ibm.com/login`
[44] `https://cloud.ibm.com/docs`
[45] `https://cloud.ibm.com/docs/tutorials/index.html`
[46] `https://cloud.ibm.com/docs/tutorials/index.html`
[47] https://cloud.ibm.com/docs/tutorials?topic=solution-tutorials-continuous-deployment-to-kubernetes
[48] `https://cloud.ibm.com/docs/java?topic=java-getting-started`

para treinamento e treinamento. Permite gerenciar dados, ativos analíticos e projetos na Cloud. (v) Aprendizado de Máquina (Machine Learning): disponibiliza frameworks de machine learning como serviços REST, tais como: TensorFlow, Keras, Caffe, PyTorch, Spark MLlib, scikit learn, xgboost and SPSS.

- Infraestrutura: bare metal server, virtual server, HPCaaS from Rescale, Power Systems Virtual Server, WebSphere Application Server, Kubernetes Service, Red Had OpenShitf Cluster, Container Registry.
- Armazenamento: Box, File Storage, IBM Cloud Backup, Object Storage, Osnexus, Portworx Enterprise.
- Banco de Dados: Bockchain Platform, Cloudant, Databases for PostgreSQL, Databases for Redis, Databases for Elasticsearch, Databases for MongoDB, Messages for RabbitMQ, Databases for etcd, Blockchain, Compose Enterprise, Compose for JanusGraph, Compose for MySQL, Db2, db2 Hosted, SQL Query etc.
- Ferramentas de desenvolvimento: Actifio Go, Availability Monitoring, Continous Delivery, Event Management etc.
- Ferramentas de rede: Load Balancers, Content Delivery Network, Direct Link Connect, Direct Link Dedicated etc.

O IBM Cloud possui um conjunto de coleções de código para iniciar ("Starter Kits") como, por exemplo, Java Microservice com Spring, Java Web App com Spring, Node.js Microservice com Express.js, Node.js Web App com Express.js, Watson Natural Language, Watson Visual Recognition etc.

Para o desenvolvimento de aplicações existem dois tipos de interface: IBM Cloud Web Console e IBM Cloud CLI (Command Line Interface ? Interface de Linha de Comando). O código desenvolvido deve ser portável, isto é:

(i) Deve estar em uma linguagem padronizada;

(ii) Deve executar na Cloud;

(iii) Deve conectar a serviços da Cloud;

(iv) Deve atender a um caso de uso específico.

Uma aplicação pode ser criada a desde o início na plataforma ou pode-se trazer código previamente implementado de uma aplicação existente para o IBM Cloud.

Em geral os passos para desenvolvimento são: (i) Criar aplicação a partir do portal do desenvolvedor; (ii) Criar uma cadeia de implantação DevOps (usando DevOps tools chain); (iii) Usar ferramentas de desenvolvimento local, i.e., suas próprias ferramentas para desenvolver; (iv) Enviar as atualizações para serem combinadas com o código no repositório remoto; (v) A implantação é feita usando a cadeia de DevOps.

Para iniciar o uso do IBM Cloud, por exemplo, execute os passos a seguir:

- Instale a ferramenta de linha de comando[49], por exemplo, a Listagem 2.14 apresenta o comando para instalar a ferramenta (Linha 1), aceitar a licença do *XCode*, caso ainda não tenha feito (Linha 3) e verificar se a instalação foi feita com sucesso (Linha 3).

- Em seguida, pode-se iniciar, criando uma aplicação (indo ao *Dashboard*, então *App* e *Create an App*) ou usando um *kit* de iniciante[50] ou a partir de uma linguagem de programação[51]. Vamos iniciar criando um microsserviço com o perfil MicroProfile do Eclipse e Java EE. Clique no link [52] e marque *Java + Liberty*. Em seguida, clique em *pattern* na caixa correspondente a este perfil.

```
1 curl -sL http://ibm.biz/idt-installer | bash
2
3 sudo xcodebuild -license accept
4
5 ibmcloud dev help
```

Listing 2.14. Comandos de configuração do ibm Cloud CLI no Mac/Linux

2.5. Prática

Esta seção descreve a parte prática a ser executada neste minicurso. Detalhes de instalação, *links* para páginas de softwares, cenário dos exercícios, exemplos de implementação, apresentações, dicas dentre outros recursos estão disponíveis na página do minicurso [Azevedo 2020]. O material está disponível no *bitbucket*[53] que é uma ferramenta Git[54] para gestão de código fonte. A seguir são apresentados informações sobre o material. Para acessar o material, o aluno pode ir diretamente na página do curso ou baixar o material localmente em seu computador usando um cliente Git.

- *Instalações*: Dicas e passo-a-passos de instalação estão disponíveis na pasta `instalacoes`. Alguns softwares a serem instalados são:
  - Git[55];
  - Maven[56];
  - Uma IDE (Integrated Development Environment) para implementação em Java, por exemplo, o NetBeans[57];
  - Docker Desktop[58];

[49] `https://cloud.ibm.com/docs/home/tools`

[50] `https://cloud.ibm.com/docs/apps/tutorials?topic=creating-apps-tutorial-starterkit`

[51] `https://cloud.ibm.com/docs/home/build`

[52] `https://cloud.ibm.com/developer/appservice/starter-kits`

[53] `https://bitbucket.org/product/`

[54] Git é um sistema de controle de versões distribuído, usado principalmente no desenvolvimento de software, mas pode ser usado para registrar o histórico de edições de qualquer tipo de arquivo.

[55] `https://git-scm.com/downloads`

[56] `a`

[57] Baixe de `https://netbeans.org/downloads/8.0.1` a versão Java EE

[58] `https://www.docker.com/products/docker-desktop`

– Postman[59] para realizar testes com os serviços implementados.

- *Cenário*: o cenário para realizar o trabalho está disponibilizado na pasta `trabalhos`. O objetivo é que o mesmo cenário seja usado para todos os exercícios e trabalhos com diferentes níveis de dificuldade.

- *Exercícios e trabalhos*: estão disponibilizados na pasta `trabalho`. O `readme` desta pasta indica a melhor ordem para aprendizado de alunos iniciantes, intermediários e avançados.

- *Materiais específicos por assunto*: as pastas `web-service`, `microservice` e `cloud` incluem apresentações, tutoriais, artigos e dicas sobre cada assunto. O `readme` de cada uma das pastas apresenta detalhes de cada um dos recursos disponibilizados nas mesmas.

2.6. Considerações Finais

Este minicurso apresentou uma visão teórica e prática de SOA, Arquitetura de Microsserviços e Cloud Computing servindo como guia prático para o aprendizado dos artefatos e ferramentas destas abordagens. Em geral estes conceitos aparecem espalhados na literatura em diversos documentos, livros, artigos, postagens de blogs etc. Neste minicurso foi dada uma visão geral, mas com nível de aprofundamento, que permite que o aluno adquira fundamentação nas três abordagens e tenha ponteiros para se aprofundar nos aspectos que mais lhe interesse.

O aprendizado destes paradigmas é essencial para o desenvolvimento de sistemas de informação modernos bem como a aplicação destes conceitos na indústria e na academia. Ainda há muitos desafios e questões a serem respondidos. No entanto, sem base teórica e prática as direções podem ser tomadas de maneira inadequada.

Dessa forma, este minicurso complementa disciplinas de graduação e pós-graduação na área de desenvolvimento de sistemas distribuídos. Ele não aborda diretamente todos os conceitos como, por exemplo, questões de desempenho, segurança, aplicação detalhada dos princípios das abordagens em diferentes domínios, os quais se constituem tópicos interessantes para serem tratados em trabalhos futuros.

Referências

[Azevedo 2020] Azevedo, L. G. (2020). Desenvolvimento de Soluções com Serviços. `https://bitbucket.org/leogazevedo/curso-soa`. Acessado em 21/02/2020.

[Azevedo et al. 2019] Azevedo, L. G., Ferreira, R. d. S., Silva, V. T. d., de Bayser, M., Soares, E. F. d. S., and Thiago, R. M. (2019). Geological Data Access on a Polyglot Database Using a Service Architecture. In *XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 103–112. ACM.

[59] `https://www.postman.com/downloads/`

[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 21/02/2020.

[Brooks Jr 1995] Brooks Jr, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Pearson Education India.

[Conway 1968] Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28–31.

[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 *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30. IEEE.

[Duan et al. 2014] Duan, Y., Narendra, N. C., Du, W., Wang, Y., and Zhou, N. (2014). Exploring Cloud Service Brokering from an Interface Perspective. In *2014 IEEE International Conference on Web Services (ICWS)*, pages 329–336. 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 2011] Fowler, M. (2011). Command Query Responsibility Segregation (CQRS). `https://martinfowler.com/bliki/CQRS.html`. Acessado em 21/02/2020.

[Fowler and Lewis 2014] Fowler, M. and Lewis, J. (2014). Microservices. `https://martinfowler.com/articles/microservices.html`. Acessado em 21/02/2020.

[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.

[Haupt et al. 2017] Haupt, F., Leymann, F., Scherer, A., and Vukojevic-Haupt, K. (2017). A framework for the structural analysis of REST APIs. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 55–58. IEEE.

[Hewitt 2009] Hewitt, E. (2009). *Java SOA Cookbook: SOA Implementation Recipes, Tips, and Techniques*. "O'Reilly Media, Inc.".

[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.

[Li and Chou 2011] Li, L. and Chou, W. (2011). Design and describe REST API without violating REST: A Petri net based approach. In *2011 IEEE International Conference on Web Services (ICWS)*, pages 508–515. IEEE.

[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.

[Mell et al. 2011] Mell, P., Grance, T., et al. (2011). The NIST Definition of Cloud computing. Technical report, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

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

[Newman 2015] Newman, S. (2015). *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, Inc.

[OpenAPI initiative 2018] OpenAPI initiative (2018). OpenAPI Specification. `https://github.com/oai/openapi-specification/blob/master/versions/3.0.1.md`.

[Papazoglou and Van Den Heuvel 2007] Papazoglou, M. P. and Van Den Heuvel, W.-J. (2007). Service Oriented Architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415.

[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 *17th International Conference on World Wide Web (WWW 2008)*, pages 805–814. ACM.

[Richards 2015] Richards, M. (2015). *Microservices vs. Service-Oriented Architecture*. O'Reilly Media.

[Richardson 2015] Richardson, C. (2015). Introduction to Microservices. `https://www.nginx.com/blog/introduction-to-microservices/`. Acessado em 21/02/2020.

[Richardson 2016] Richardson, C. (2016). Microservice Architecture Patterns and Best Practices. `http://microservices.io`. Acessado em 21/02/2020.

[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.

[Santos et al. 2020] Santos, J. S., Azevedo, L. G., Soares, E. F. S., Thiago, R. M., and Silva, V. T. (2020). Analysis of Tools for REST Contract Specification in Swagger/OpenAPI. In *2nd International Conference on Enterprise Information Systems (ICEIS 2020)*. INSTICC.

[Tsouroplis et al. 2015] Tsouroplis, R., Petychakis, M., Alvertis, I., Biliri, E., and Askounis, D. (2015). Community-based API Builder to manage APIs and their connections with Cloud-based Services. In *CAiSE Forum*, pages 17–23.

[Varanasi and Belida 2015] Varanasi, B. and Belida, S. (2015). *Introduction to REST*, pages 1–13. Apress, Berkeley, CA.

[Vernon 2013] Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.

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

[Villaça et al. 2020] Villaça, L. H., Azevedo, L. G., and Moreno, M. (2020). Microservice Architecture for Multistore Database Using Canonical Data Model. In *Proceedings of the XIV Brazilian Symposium on Software Components, Architectures, and Reuse*. ACM.

[Villaça et al. 2018b] Villaça, L. H., Pimenta Jr., A. F., and Azevedo, L. G. (2018b). Construindo aplicações distribuídas com microsserviços. In *Tópicos em Sistemas de Informação: Minicursos XV Simpósio Brasileiro de Sistemas de Informação*. SBC.

[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 2017] Zimmermann, O. (2017). Microservices tenets. *Computer Science-Research and Development*, 32(3-4):301–310.

Biografia do autor

Leonardo G. Azevedo é pesquisador da IBM Research Brazil desde 2013. Ele é Doutor (2005) e Mestre (2001) pelo PESC/COPPE/UFRJ e Bacharel em Informática (2000) pela UFRJ. Ele foi professor da UniRio de 2006 a 2018, lecionando disciplinas em Banco de Dados e Engenharia de Software e atuou no Programa de Pós-Graduação em Informática (PPGI/UNIRIO). Leonardo tem mais de 20 anos de experiência em pesquisa e desenvolvimento de sistemas e vem atuando em projetos para diferentes empresas nacionais e internacionais e governo. Suas áreas de pesquisa incluem Sistemas Distribuídos, Arquitetura Orientada a Serviços (SOA), Microsserviços, Representação do Conhecimento, Ontologias, Gestão de Processos de Negócio (BPM), Computação Cognitiva, e Banco de Dados Espaciais. Para maiores detalhes http://researcher.ibm.com/researcher/view.php?person=br-lga e http://lattes.cnpq.br/7214791464543522.