UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SAMIR BRAGA CHAVES

SERVICER: UMA PLATAFORMA PARA O DESENVOLVIMENTO RÁPIDO DE MICROSSERVIÇOS MULTILINGUAGEM

FORTALEZA

2021

SAMIR BRAGA CHAVES

SERVICER: UMA PLATAFORMA PARA O DESENVOLVIMENTO RÁPIDO DE MICROSSERVIÇOS MULTILINGUAGEM

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. José Antônio Macêdo

Coorientador: Prof. Dr. Regis Pires Magalhães

FORTALEZA

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C439s Chaves, Samir Braga.
Servicer: Uma Plataforma para o Desenvolvimento Rápido de Microsserviços Multilinguagem / Samir Braga Chaves. – 2021.
58 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Ciências, Curso de Computação, Fortaleza, 2021.
Orientação: Prof. Dr. José Antônio Macêdo.
Coorientação: Prof. Dr. Regis Pires Magalhães.

1. Arquitetura de microsserviços. 2. Framework de microsserviços. 3. Sistemas multilinguagem. 4. Integração. 5. Definição de microsserviços. I. Título.

CDD 005

SAMIR BRAGA CHAVES

SERVICER: UMA PLATAFORMA PARA O DESENVOLVIMENTO RÁPIDO DE MICROSSERVIÇOS MULTILINGUAGEM

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. José Antônio Macêdo (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Regis Pires Magalhães (Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Davi Romero de Vasconcelos
Universidade Federal do Ceará (UFC)

Ao meus pais, irmã e namorada, por sua capacidade de acreditar e investir em mim. Nada seria sem seus ensinamentos e apoio. A base forte que hoje me sustenta foi por vocês alicerçada.

AGRADECIMENTOS

Aos professores Dr. Regis Pires Magalhães e Dr. José Antônio Fernandes de Macêdo pela orientação no presente trabalho, pelas dicas e conselhos essenciais, pelas reuniões de acompanhamento e pelas conversas descontraídas.

Aos meus pais que sempre me apoiaram e investiram na minha educação como puderam, independente de todas as dificuldades. À minha irmã, que me motivou a manter o interesse nas áreas além das exatas. E à minha namorada, pela sua paciência, capacidade de me entender e seus conselhos.

Aos meus colegas de graduação que partilharam as dificuldades e conquistas dessa longa trajetória. Em especial, ao Franklyn Seabra e Jarelio Gomes que escreveram suas monografias em concomitância com a minha e dos quais obtive valiosas ajudas.

Ao Insight Data Science Lab, onde trabalho e pesquiso na data em que escrevo, uma das maiores fontes de conhecimento que já experimentei, onde fiz amigos inigualáveis e tive experiências enriquecedoras tanto na pesquisa quanto na vida.

A todo o Departamento de Computação, sua coordenação e secretaria, das quais o suporte foi decisivo durante toda a graduação. E, especialmente, ao seu corpo docente pelo aprendizado proporcionado e por ter mostrado o valor e a força da ciência.

"Se chegássemos ao fim da linha, o espírito humano feneceria e morreria. Mas acho que nunca vamos ficar estagnados: devemos crescer em complexidade, quando não em profundidade, e seremos sempre o centro de um horizonte de possibilidades em expansão."

(Stephen Hawking)

RESUMO

A adoção da arquitetura de microsserviços vem crescendo nos últimos anos devido às suas múltiplas vantagens. No entanto, esses benefícios vêm acompanhados de diversos desafios durante o desenvolvimento. Tais desafios, apesar de já contarem com várias soluções, ainda são tratados de maneiras diferentes por cada *framework*, os quais são, na grande maioria dos casos, restritos apenas uma linguagem. Com base neste cenário, o presente trabalho apresenta o Servicer, uma plataforma que busca facilitar o desenvolvimento de microsserviços ao passo que automatiza processos da integração entre diferentes linguagens de programação. O Servicer é baseado em uma arquitetura geral e foi implementado em um conjunto específico de tecnologias que estão no estado da arte dos padrões e conceitos abordados na versão apresentada. Além disso, é desenvolvido um caso de uso visando avaliar os principais aspectos da arquitetura e implementação da plataforma, a partir do qual percebeu-se grande ganho de produtividade no desenvolvimento relacionado ao tempo de criação e integração de microsserviços multilinguagem à medida que observou-se algumas limitações.

Palavras-chave: Arquitetura de Microsserviços. *Framework* de microsserviços. Sistemas multilinguagem. Integração. Definição de Microsserviços. Sistemas distribuídos.

ABSTRACT

The adoption of the microservice architecture has been growing in recent years due to its multiple advantages. However, these benefits come with several challenges during development. Such challenges, despite already having several solutions, are still treated in different ways by each framework, which are, in the vast majority of cases, restricted to just one language. Based on this scenario, the present work presents Servicer, a platform that aims to facilitate the development of microservices while automating integration processes between different programming languages. The Servicer is based on a general architecture and has been implemented in a specific set of technologies that are in the state of the art of the standards and concepts covered in the presented version. In addition, a use case is developed in order to evaluate the main aspects of the platform architecture and implementation, from which a great productivity gain was noticed in the development related to the time of creation and integration of multilanguage microservices as was observed some limitations.

Palavras-chave: Microservice architecture. Microservice framework. Multilanguage systems. Integration. Microservice definition. Distributed Systems.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

MS	Microsserviços
JSON	*Javascript Object Notation*
SOA	*Service-oriented architecture*
MSC	*Microservice Chassis*
CLI	*Command Line Interface*
DSL	*Domain Specific Language*
DTO	*Data Transfer Object*
ML	*Machine Learning*
CIDE	CAOPLE Integrated Development Environment
JVM	*Java Virtual Machine*
NPM	*Node Package Manager*
SBT	*Scala Build Tool*
SDK	*Software Development Kit*
JWT	*JSON Web Token*

SUMÁRIO

1 INTRODUÇÃO

Temas como escalabilidade, computação em nuvem, segurança, integração e monitoramento estão cada vez mais presentes nos ambientes organizacionais quando o assunto é criação e gerenciamento de aplicações. Isso faz com que seja criada, tanto no âmbito acadêmico quanto mercadológico, uma demanda por soluções que abordem essas questões e ataquem os seus respectivos desafios de forma especializada (KLEPPMANN, 2018).

Com essa problemática em vista, vários serviços são apresentados visando solucioná-la, disponibilizados de maneira prática e rápida, buscando atender as necessidades de uma sociedade com exigências cada vez mais complexas. Tais necessidades são assumidas por instituições públicas e privadas que, ao trabalhar com aplicações, buscam entregar valor aos usuários ao passo que também buscam se estabelecer em um contexto de ampla concorrência que é o da TI. Portanto, é clara a necessidade de ferramentas que permitam um maior tempo dedicado ao desenvolvimento das lógicas do serviço a ser criado, ao invés de tempo gasto com problemas oriundos da escolha de uma arquitetura, tecnologia, protocolo ou provedor de algum serviço.

A adoção da arquitetura de microsserviços vem crescendo nos últimos anos (FRANCESCO *et al.*, 2017) por possuir diversas vantagens relacionadas tanto à características técnicas do desenvolvimento e gerenciamento, quanto em relação ao custo, já que, como é mostrado no trabalho de Villamizar *et al.* (2016), o uso de microsserviços pode reduzir custos com infraestrutura em 70% ou mais. No entanto, esses benefícios têm um custo relacionado aos diversos desafios que essa arquitetura traz consigo, os quais são descritos na seção 2.2.

Muitas soluções na forma de linguagens, *frameworks* ou padrões visam atacar esses desafios abstraindo algumas das suas complexidades e tornando mais práticos processos como desenvolvimento, monitoramento e *deploy* das aplicações. Entretanto, tratando-se mais especificamente do desenvolvimento, apesar de uma das principais características da arquitetura de microsserviços ser a independência entre os serviços (NEWMAN, 2019), muitas dessas ferramentas provém funcionalidades restritas a alguma linguagem específica, deixando a cargo do desenvolvedor a integração com outras linguagens, se necessário. Além disso, cada *framework* tem sua própria maneira de lidar com determinados problemas, tornando ainda mais difícil essa integração.

Situado nesse contexto, o presente trabalho propõe uma solução que busca tornar o desenvolvimento de microsserviços e a sua integração em diferentes linguagens uma tarefa

mais simples e rápida, garantindo flexibilidade e customização. Isso se dá por meio da proposta de uma interface comum para a definição de microsserviços e de uma plataforma que busca automatizar processos relacionados à criação de uma aplicação multilinguagem distribuída.

1.1 Objetivos

A seguir, os objetivos geral e específicos do presente trabalho.

1.1.1 Objetivo Geral

Desenvolver uma plataforma que possibilite o desenvolvimento e integração de Microsserviços (MS) em diferentes linguagens, de fácil integração com tecnologias já existentes, além de permitir a adição de novas funcionalidades por meio de extensões.

1.1.2 Objetivo Específicos

- Elaborar uma plataforma modular e extensível que permita a integração de módulos escritos em diferentes linguagens.
- Criar um componente central responsável por gerenciar a aplicação e seus diferentes módulos.
- Especificar um conjunto mínimo de definições necessárias para expressar um MS em alto nível e independente de linguagem.
- Implementar casos de uso dessa plataforma com um exemplo de aplicação real.

Os seguintes capítulos estão dispostos como segue: é abordada a fundamentação teórica dos principais conceitos utilizados durante todo o trabalho no Capítulo 2. No Capítulo 3, são apresentados e comparados trabalhos relacionados à pesquisa da presente monografia e da solução nela proposta. Já no Capítulo 4, é descrita a metodologia elaborada e seguida durante a pesquisa e durante o desenvolvimento da solução. A descrição da solução, envolvendo requisitos, arquitetura e implementação é trazida no Capítulo 5. No Capítulo 6, são explicados os casos de uso da solução. Por fim, as conclusões e trabalhos futuros estão presentes no Capítulo 7.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados conceitos importantes na compreensão dos capítulos posteriores.

2.1 REST API

APIs REST são mecanismos de comunicação entre processos, quase sempre aplicados sobre o protocolo HTTP, e que seguem o estilo arquitetural *Representational State Transfer* (REST) (RICHARDSON, 2019). Essa API provê operações que manipulam um determinado recurso. Esse recurso geralmente se apresenta como um objeto ou uma coleção de objetos de um domínio de negócio, acessível por uma URL e cuja representação pode ser, por exemplo, em XML, *Javascript Object Notation* (JSON) ou binário. Além disso, essas operações são atribuídas semanticamente a verbos do HTTP: GET, POST, PUT, DELETE, etc.

Um outro conceito frequentemente apresentado nesse contexto é o de APIs RESTFul. O conceito de REST é definido por Fielding (2000), onde trata-se de um padrão agnóstico ao protocolo, entretanto, não há um consenso sobre um conceito formal de RESTFul. Como é explicado no *Richardson Maturity Model* (FOWLER, 2010), considera-se que este está relacionado a como os recursos do HTTP são explorados semanticamente e como a API pode ser auto-documentada.

2.1.1 Benefícios

Por se basearem simplesmente no HTTP, as APIs RESTFul são bem simples de se compreender e implementar, assim como são acessíveis por diversas ferramentas que permitem esse tipo de comunicação, como o próprio navegador. Além disso, o modelo de comunicação síncrono de requisição e resposta representa muito bem a lógica de muitos sistemas.

2.1.2 Desvantagens

- Há alguns sistemas que precisam de uma comunicação assíncrona, a qual não é atendida naturalmente pelas APIs REST.
- Como as operações são acessíveis por URLs, estas devem ser previamente conhecidas exatamente como são em todos os aspectos necessários (parâmetros da URL, corpo da

requisição, corpo e *status* da resposta ou *headers*).

– Nem sempre a tradução das operações de manipulação para os verbos HTTP é natural.

2.2 Microsserviços

Microsserviços é um estilo de arquitetura orientada a serviços *Service-oriented architecture (SOA)*, com a vantagem de que os serviços são agnósticos em tecnologia (NEWMAN, 2019). SOA estrutura a visualização da implementação como um conjunto de vários componentes (serviços) conectados por protocolos de comunicação que permitem que eles colaborem entre si(RICHARDSON, 2019). Sua adoção vem crescendo bastante nos últimos anos, não somente por empresas relacionadas a software mas também por outras que simplesmente necessitam de uma arquitetura que atenda as suas altas demandas (BALALAIE *et al.*, 2016).

Principais características dos microsserviços:

– Disponibilizados de forma independente um dos outros.

– São modelados em torno de um único domínio de negócio.

– Não há compartilhamento entre seus bancos de dados.

– Comunicam-se por meio da rede.

Os serviços trocam informações por meio de mecanismos de comunicação entre processos, como *APIs REST* e mensagens assíncronas (RICHARDSON, 2019). Logo, é nessa comunicação entre os microsserviços que as questões sobre *APIs REST* apresentadas estão presentes, por exemplo, a comunicação seria síncrona, daria-se sobre o protocolo HTTP e havendo manipulação de recursos.

2.2.1 Vantagens

Todas as características apresentadas promovem flexibilidade. Como os serviços são independentes em vários aspectos, isso permite que atualizações em um serviço não tenham consequências em outro, assim como o *deploy* dessa modificação não envolva todo o sistema e sim apenas uma pequena parte dele. Além disso, devido ao baixo acoplamento, diversos aspectos podem ser facilmente paralelizados e assim permitir uma maior escalabilidade. Um deles é o desenvolvimento, já que cada serviço pode ser encarado como um "microssistema"com escopo bem determinado e ainda com tecnologias e estilos possivelmente diferentes. Outro é a própria execução, a garantia que o bom funcionamento se permanecerá ao longo dos vários cenários,

nos quais aumentam-se, por exemplo, o número de requisições (KLEPPMANN, 2018).

2.2.2 Desafios Inerentes

A independência que justifica grande parte das vantagens também causa alguns problemas, muitos deles oriundos da comunicação pela rede e da característica distribuída da arquitetura. Problemas como latência, falhas na comunicação, *timeout* e acesso às várias instâncias dos vários serviços requerem técnicas especializadas em lidar com cada uma delas e assim proporcionar um bom funcionamento. Visando formalizar o tratamento dessas problemáticas surgiram muitos padrões, geralmente complementares, que podem ser agrupados de acordo com o contexto do problema. Alguns desses padrões, juntamente com os problemas que os motivaram, e que são aprofundados em Richardson (2019):

1. *Descoberta de Serviços*: em um ambiente de nuvem, os serviços podem possuir mais de uma instância em execução, as quais possuem diferentes endereços de rede. Essas instâncias podem ser encerradas (por diversos motivos, como falhas nos processos internos ou simplesmente por não haver carga de trabalho que justifique a permanência de sua execução) e recriadas constantemente. Assim, é necessário manter um registro das instâncias em execução dos diferentes serviços e seus respectivos endereços. A descoberta de serviços se trata do processo de resolver a comunicação entre os serviços, independente de suas localizações na rede.
2. *Health Check API*: as plataformas de *deploy* precisam conhecer quais instâncias estão indisponíveis para evitar o direcionamento de requisições a elas e posteriormente terminá-las, caso não se recuperem. Essa informação é provida pelo padrão de observabilidade *Health Check*, que possibilita verificar a disponibilidade de um serviço. Uma implementação comum se baseia em oferecer uma funcionalidade na API do serviço que verifique a integridade de todos os recursos dos quais depende, como bancos de dados, serviços externos e etc.
3. *API Gateway*: existem regras de acesso a um serviço que não necessitam serem gerenciadas por ele, como autenticação e autorização. Além disso, diferentes tipos de clientes podem receber diferentes tratamentos das respostas dos serviços. Por fim, o acesso direto aos serviços dentro da sua infraestrutura pode ter consequências relacionadas à segurança. Por isso, o padrão API Gateway se baseia em manter um único ponto de entrada para os serviços da infraestrutura. Este ponto de entrada é responsável por realizar o roteamento das

requisições, tradução de protocolos e implementar algumas operações sobre as respostas dos serviços, como o *API Composition*[1]. Além disso, outra responsabilidade importante é a implementação de *edge functions*, como autenticação, autorização, coleta de métricas, *logging* das requisições, entre outras.

4. *Circuit Breaker*: relacionado ao problema de comunicações confiáveis, o *Circuit Breaker* é um padrão que busca tratar falhas parciais nos serviços. Assim, são contabilizadas as quantidades de requisições com sucesso e com falhas. Caso a taxa de erro supere um limiar predefinido, o *circuit breaker* será iniciado, fazendo com que as requisições falhem imediatamente. Após um intervalo de tempo também predefinido, é verificado se já é possível voltar a encaminhar as mensagens para o serviço, se sim o *circuit breaker* é encerrado e o fluxo volta à normalidade.

2.2.3 Microservice Chassis

Dados os desafios apresentados, algumas ferramentas buscam resolvê-lo de maneira geral abstraindo-os do desenvolvedor. Geralmente, essas soluções apresentam-se na forma de *frameworks* em diversas linguagens como o Spring Boot e Spring Cloud (Java e Kotlin), Gizmo e Micro (GO), Lagom (Scala e Java), Moleculer.js (Node.js), entre outros. A esse conjunto de tecnologias especializadas a resolver os desafios inerentes à arquitetura de microsserviços e evitar um retrabalho por parte da equipe de desenvolvimento, dá-se o nome de *Microservice Chassis* (MSC) (RICHARDSON, 2019).

[1] https://microservices.io/patterns/data/api-composition.html

3 TRABALHOS RELACIONADOS

Nesta sessão serão apresentadas como trabalhos relacionados três ferramentas que possuem intuitos semelhantes à criada na atual dissertação. Tais semelhanças estão mais relacionadas à sua filosofia, propósito e resultados do que à suas particulares técnicas. A característica principal para a seleção foi o propósito de auxiliar na criação de aplicações orientadas à serviços. São elas:

3.1 JHipster

Surgido em 2013, JHipster é uma plataforma que busca facilitar o desenvolvimento de aplicações modernas utilizando Javascript e Java (HALIN *et al.*, 2017). Ele se dá por meio de um *Command Line Interface* (CLI) que possibilita a geração de códigos relacionados tanto ao *frontend* quando ao *backend*. De acordo com estatísticas oficiais, no fim de 2019 já eram mais de 20.000 aplicações geradas por mês e em torno de 2.500.000 instalações desde de sua disponibilização, além de ter recebido contribuições oficiais do Google, Microsoft, Red Hat, Heroku e etc (SASIDHARAN, 2020).

Com essa ferramenta é possível desenvolver, testar e disponibilizar aplicações em diferentes ambientes de nuvem. Com relação ao *frontend*, possui integração com React.js, Angular e Vue.js, três dos mais conhecidos *frameworks* Javascript para criação de interfaces web. Já se tratando de *backend*, oferece ao desenvolvedor a possibilidade de escolha: gerar uma aplicação monolítica ou orientada a microsserviços. Estas possibilidades combinadas suprem a necessidade de uma boa parte das aplicações que se integram bem com essa *stack* de tecnologias.

3.1.1 Principais Vantagens

Na visão do autor, a principal vantagem do JHipster é a abstração oferecida sobre o processo de configuração inicial das diversas tecnologias envolvidas em uma aplicação mais complexa. Esse processo é, muitas vezes, uma tarefa árdua, pois envolve, além do desenvolvimento do código inicial, a integração dos diferentes componentes e suas dependências de forma correta. Somado a isso, as tecnologias envolvidas já estão amplamente difundidas e possuem uma forte comunidade.

A solução também possui uma *Domain Specific Language* (DSL) chamada JHispter Domain Language (JDL), que permite a definição de diversos componentes. Um destes é o

conjunto de modelos de entidades e seus relacionamentos que servem para a criação de *Data Transfer Object* (DTO) dentro da aplicação usados diretamente nos *endpoints* REST e para sua integração com a camada de serviço. A JDL também conta com uma ferramenta visual chamada JDL-Studio[1] que permite visualizar as entidades e suas relações em diagramas UML.

Com foco em microsserviços, nessa mesma linguagem é possível também definir as próprias aplicações com configurações como nome, *build tool*, tipo da aplicação, mecanismos de descoberta de serviço, mecanismo de autenticação, sistema de bancos de dados a ser utilizado, entre outras. Com isso, é permitido ao desenvolvedor definir algumas aplicações do tipo *microservice* com suas respectivas entidades e uma aplicação do tipo *gateway* e, assim, ele terá criado um servidor orientado a dados e distribuído em poucos passos (SASIDHARAN, 2020).

3.1.2 Limitações

A abstração citada na seção anterior é de grande valia para uma equipe que já possui domínio sobre os artefatos gerados pela ferramenta, pois, a partir deles, é possível estender a aplicação e sua complexidade até o nível desejado no projeto. Entretanto, a quantidade de código gerado no *bootstrap*, a semântica de cada componente e as suas integrações já são complexos o suficiente para quem não possuem esse domínio. Equipes com menor maturidade no conjunto de tecnologias provido terá uma dificuldade semelhante à existente no contexto sem uso do JHipster quando houver a necessidade de customizar as lógicas de negócio.

Um outro ponto que na visão do autor dificulta a aplicação do JHipster em alguns cenários é o fato de, além de trabalhar apenas com Java, suportar apenas projetos do Spring (HALIN *et al.*, 2019). Outras tecnologias deverão ser integradas manualmente, como Python - com uso abrangente em aprendizagem de máquina e análise de dados, Scala - para processamento de *streams* e programação naturalmente reativa para servidores e ainda Node.js - com desenvolvimento rápido e com APIs práticas para aplicações em tempo real. Este foi um dos principais motivos que incitou a criação do presente trabalho.

Vale ressaltar que os pontos acima apresentados não são defeitos, apenas consequências das escolhas feitas pela equipe de desenvolvimento da ferramenta e que estão sendo analisadas em um contexto real para os diferentes cenários de uso.

[1] https://www.jhipster.tech/jdl-studio/

3.2 Cortex

Cortex é uma solução de código aberto criada para disponibilizar modelos de aprendizagem de máquina em ambientes de produção seguindo uma estratégia *Serveless*. Com sua primeira versão publicada em fevereiro de 2019, a ferramenta possui integração com as principais bibliotecas do tema, como PyTorch[2], Keras[3], Scikit-learn[4] e TensorFlow[5] (LABS, 2021).

A intenção dos criadores é que o desenvolvedor, após ter seu modelo criado, apenas informe como ocorrerá a entrada e saída de dados e, com isso, a plataforma se responsabiliza por criar uma API REST para que essa comunicação ocorra pela rede e disponibilizá-la em produção. Existem duas possibilidades para o *deploy*: ou em uma plataforma de nuvem como a Amazon Web Services e Google Cloud Platform usando o Cortex Cloud ou em uma infraestrutura própria sobre o Kubernetes usando o Cortex Core.

3.2.1 Vantagens

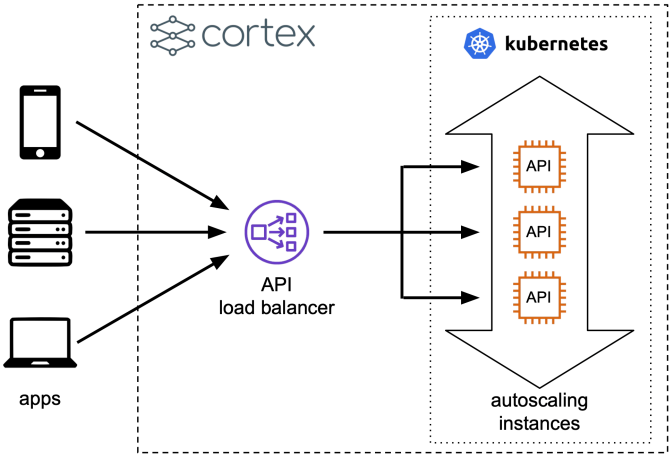
A principal vantagem do Cortex, que inclusive o levou a estar presente neste trabalho e serviu como inspiração, é a abstração sobre a característica distribuída da aplicação que é criada, assim como sobre sua camada de acesso pela rede. Ao focar no problema específico de *Machine Learning* (ML) em Python, conseguiu resolver esses desafios de uma maneira prática e funcional, permitindo que o usuário foque apenas na sua lógica de negócio. Apesar de não explicitar o uso da arquitetura de microsserviços, é possível notá-la em muitos aspectos, como:

- A possibilidade de que os serviços responsáveis por executar os modelos conversem entre si e provenham uma predição integrada por meio de *Chaining APIs*[6].
- As requisições feitas seguindo o *Chaining APIs* são resolvidas por uma estratégia de descoberta de serviços.
- A arquitetura de rede da API disponibilizada para acesso aos modelos, representada na Figura 1, segue outros padrões como *API Gateway*, *Load Balancing* e *Autoscaling*, todos comuns no ambiente de microsserviços (RICHARDSON, 2019).
- Como o *deploy* é realizado pela plataforma, esta já é capaz de integrar-se com ferramentas de monitoria enviando logs e métricas sobre os modelos e sobre as instâncias em que estão

[2] https://pytorch.org/
[3] https://keras.io/
[4] https://scikit-learn.org/
[5] https://www.tensorflow.org/
[6] https://docs.cortex.dev/workloads/realtime-apis/predictors#chaining-apis

presentes.

Figura 1 – Diagrama da arquitetura de rede do Cortex.



Fonte: https://docs.cortex.dev/clusters/cortex-cloud-on-aws/index

3.2.2 Limitações

- Focado apenas no domínio de aprendizagem de máquina.
- Disponível apenas em Python.
- Poucas possibilidades de customização da API REST gerada.

3.3 CIDE

Neste trabalho de Liu *et al.* (2016) é apresentada o CAOPLE Integrated Development Environment (CIDE), uma proposta de ambiente para o desenvolvimento de microsserviços baseada na linguagem CAOPLE, a qual foi apresentada em outro trabalho de Xu *et al.* (2016), um dos autores do primeiro. Este ambiente traz um conjunto de ferramentas que auxiliam nos processos de desenvolvimento, *deploy* e monitoramento de serviços, além de ferramentas para gerenciamento dos recursos de computação em nuvem.

3.3.1 CAOPLE

Como é explicada em ambas as publicações, CAOPLE é uma linguagem orientada a agentes, centrada em castas e criada com o propósito de trazer consigo soluções para os desafios existentes no desenvolvimento de microsserviços. Tais soluções são consequências do seu modelo conceitual constituído por duas entidades: **classes** e **agentes**.

Um Agente é entidade computacional que provê a funcionalidade de um serviço. Já uma classe é definida como um classificador de agentes, ou seja, agentes são instâncias em execução de classes, assim como objetos são instâncias de classes em um paradigma orientado a objetos. A CAOPLE é executada sobre uma máquina virtual chamada CAVM-2, a qual é formada por duas entidades: *Communication Engine* (CE) responsável por uma comunicação sobre a rede transparente entre os agentes e o *Local Execution Engine* (LEE) que executa o código do programa de um agente.

3.3.2 Benefícios

Um objetivo em comum entre o CIDE e o presente trabalho é permitir a definição de *MS* em um alto nível de abstração. O CIDE lida com isso permitindo a construção de modelos gráficos de sistemas em uma linguagem chamada CAMLE (SHAN; ZHU, 2004). Tais modelos são convertidos para especificações formais e depois transformados automaticamente em código CAOPLE, permitindo assim uma redução da complexidade do desenvolvimento. Outro benefício é transparência na comunicação entre os agentes oferecido pelo CE, que funciona como um centro de distribuição de mensagens permitindo que os serviços enviem e recebam dados de forma assíncrona.

3.3.3 Limitações

O CIDE oferece um conjunto de ferramentas integradas que lidam bem com os desafios de MS, no entanto, não foi apresentada possibilidade de integração com outras tecnologias e linguagens já presentes no mercado. Sendo assim, apesar de boa parte das tarefas presentes no ciclo de desenvolvimento, teste, disponibilização e monitoramento já terem sido resolvidas pelo ambiente apresentado, o seu usuário está restrito a trabalhar apenas com as ferramentas a ele providas nesse ambiente.

3.4 Comparativo e Avaliações Finais

Nesta seção é trazida no Quadro 1 uma comparação entre os trabalhos explanadas neste capítulo e o Servicer, solução apresentada na atual dissertação e destacada em negrito. A comparação foi realizada sobre 6 critérios, os quais serão apresentados a seguir. A presença do símbolo ($\times$) em algum dos valores significa a não aplicação daquele trabalho no critério

avaliado.

O primeiro critério é referente à *'geração de código fonte e configurações'*. É importante ressaltar que, neste trabalho, entende-se essa geração como a construção de artefatos que serão gerenciados pelo desenvolvedor nos passos seguintes as suas criações. Logo, a geração de código como um processo interno, sem interferência do usuário, não é considerada nesse critério. Os dois principais processos onde é avaliada a necessidade dessa geração são a definição de um MS e construção dos modelos de entidades compartilhados entres os serviços.

Como segundo critério, tem-se o *'domínio'* cujo significado está relacionado ao propósito da solução analisada. Por exemplo, o objetivo do JHipster é a criação tanto de monólitos quanto de microsserviços, enquanto o do CIDE é apenas de microsserviços.

A próxima característica avaliada trata-se das *'linguagens suportadas'* pela ferramenta no desenvolvimento de MS e suas lógicas de negócios. No caso do Servicer, sua proposta é integrá-lo à qualquer linguagem já existente, bastando apenas que essa suporte a escrita do módulo necessário para essa integração, apresentado na subseção 5.2.1.2. Devido a isso, o seu valor nessa categoria é "Linguagens integradas".

A penúltima avaliação é sobre o suporte da ferramenta à mais de uma linguagem de programação para a criação de conjunto misto de MS.

Por fim, a sexta característica refere-se a capacidade da solução de prover uma integração automática do serviços externos ao ambiente já existente em um *framework* específico.

Portanto, objetiva-se neste trabalho elaborar uma proposta de ferramenta que auxilie no desenvolvimento de microsserviços oferecendo o suporte a diferentes linguagens e *frameworks*, possibilitando assim a rápida integração e comunicação entre MS em um ambiente misto. A solução foi denominada Servicer e com ela também é possível estender suas funcionalidades através de *plugins* como será demonstrado no Capítulo 5.

Quadro 1 – Comparativo dos trabalhos relacionados e solução proposta.

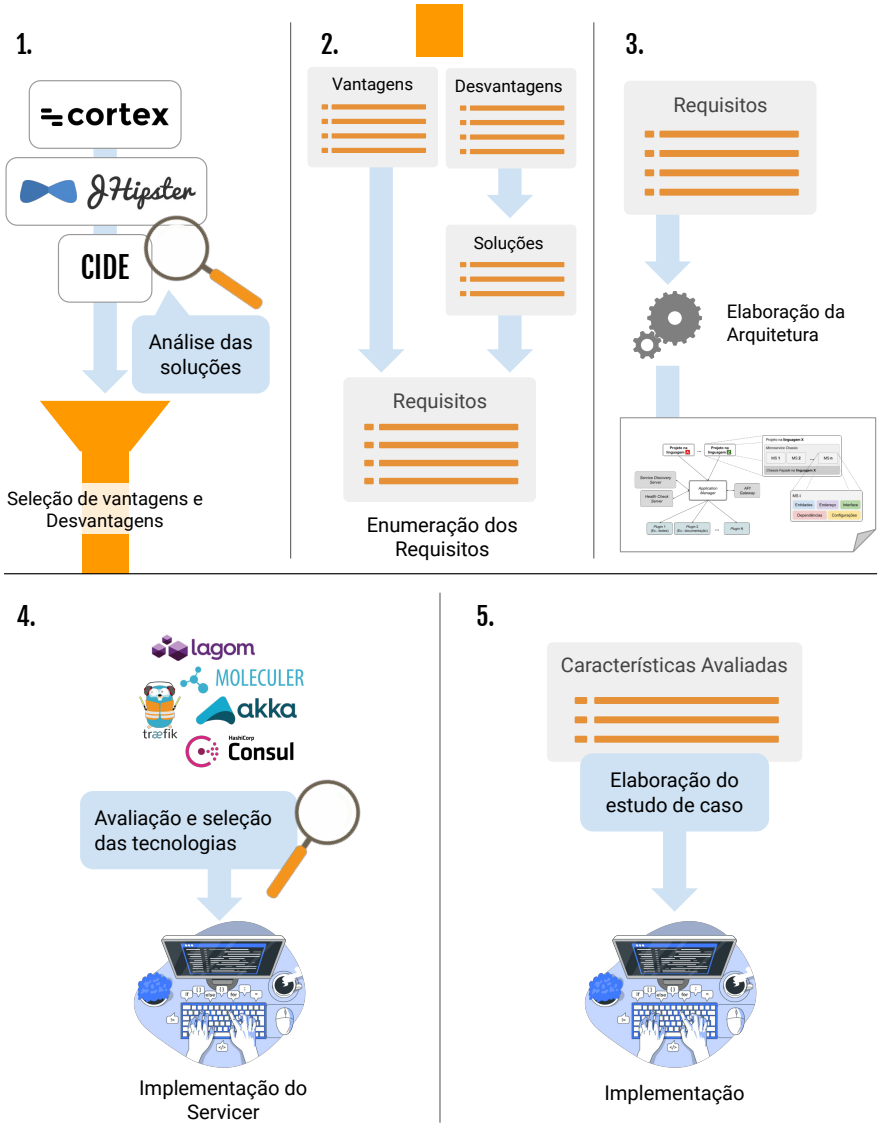
	JHipster	Cortex	CIDE	Servicer
Geração de código fonte e configurações	Sim	Não	Sim	Não
Domínio	Monólitos e MS	Modelos de ML	MS	MS
Linguagens suportadas	Java e Kotlin	Python	CAOPLE	Linguagens integradas
Multilinguagem	Não	Não	Não	Sim
Integração automática de serviços externos	Não	Não	Não	Sim

Fonte: Elaborado pelo autor

4 METODOLOGIA

As seções seguintes detalharão a metodologia seguida neste trabalho, a qual foi dividida em 5 etapas, ilustradas na Figura 2: análise das soluções existentes para criação de microsserviços, enumeração dos requisitos da solução proposta, elaboração da arquitetura da solução, escolha das tecnologias envolvidas na solução e sua implementação e, por fim, elaboração e implementação dos casos de uso.

Figura 2 – Passos seguidos na metodologia do trabalho.



Fonte: Elaborado pelo autor
Descrição: (1) São avaliadas as três soluções apresentadas e, a partir disso, selecionadas suas vantagens e desvantagens; (2) Com a análise das soluções, foram enumerados os requisitos do Servicer a partir do que é considerado positivo nas demais propostas e de alternativas para o que foi considerado negativo; (3) A partir dos requisitos foi elaborado uma arquitetura geral buscando atendê-los; (4) Em posse da arquitetura, foram avaliadas e selecionadas ferramentas para implementar uma versão funcional; (5) Buscando avaliar as características do Servicer, foi elaborado e implementado um estudo de caso.

4.1 Análise das Soluções Existentes para Criação de Microsserviços

Como preparação para a próxima etapa da metodologia, esta traz os prós e contras encontrados nas três soluções abordadas no Capítulo 3. Como o foco inicial do Servicer é a etapa de desenvolvimento, as características avaliadas são voltadas apenas a isso, considerando como trabalhos futuros à análise e implementação de funcionalidades relacionadas à etapa de produção.

4.1.1 Vantagens

Segue abaixo a lista com a união das características presentes no JHipster, Cortex e CIDE e consideradas acertos a serem mantidos e adaptados.

- Definição de um MS em alto nível;
- Integração com *frameworks* existentes;
- Integração com ferramentas auxiliares já bem estabelecidas no contexto de MS;
- Comunicação transparente entre os microsserviços, tanto em relação às suas localizações na rede quanto à estrutura dos dados transferidos nas mensagens.

4.1.2 Desvantagens

Já na listagem a seguir são apresentadas características tidas como limitantes quando se necessita de um conjunto misto de microsserviços.

- Geração de código para definição dos microsserviços;
- Escopo restrito;
- Limitação apenas a uma linguagem ou *framework*.

4.2 Enumeração dos Requisitos da Solução Proposta

Dadas as características positivas e negativas discriminadas na seção anterior, o conjunto de requisitos da plataforma aqui proposta será formado a partir da permanência e adaptação do que já satisfaz as demandas existentes no contexto de MS somados à soluções para o que observou-se ser limitante nas propostas analisadas.

Por exemplo, para o problema de geração de código, pretende-se ler a definição do MS e criá-lo em tempo de execução, quando possível, sem gerar artefatos que necessitem ser

gerenciados pelo desenvolvedor. Já em relação ao escopo, objetiva-se não restringi-lo e permitir a criação de aplicações de propósito geral. Por fim, sobre a limitação à apenas linguagem ou *framework* será criada uma arquitetura que reduza os obstáculos na integração desses diferentes sistemas.

Uma das principais vantagens dessa arquitetura é a flexibilidade em se valer das várias tecnologias disponíveis para prover serviços independentes e suas atualizações de forma rápida e eficaz (NEWMAN, 2019). O uso de diferentes linguagens de programação é o que provê boa parte dessa flexibilidade ao permitir que a equipe tenha liberdade em escolher a linguagem mais adequada para a implementação da lógica inerente ao domínio de cada serviço.

Ao desenvolver microsserviços em um ambiente com múltiplas linguagens, alguns desafios surgem durante o processo de integração. Cada serviço não pertencente ao conjunto de serviços definido em um *framework* é por ele considerado um **serviço externo** e assim há a necessidade de defini-lo manualmente nos padrões esperados por essa ferramenta. Pretende-se automatizar grande parte desses processos.

4.3 Elaboração Arquitetura da Solução

Com base nos requisitos elucidados, foi elaborada uma arquitetura geral que oferece uma melhor integração entre as diferentes linguagens e tecnologias, principal deficiência observada nos trabalhos analisados. A fim de tornar tal arquitetura compreensível e consistente, esta foi constituída por diferentes componentes, os quais possuem responsabilidades bem definidas e que buscam aplicar conceitos já bem estabelecidos no contexto de microsserviços. Assim, apresentados seus papeis, implementar cada componente em diferentes ferramentas não deve ser uma tarefa árdua, devido à característica modular da arquitetura.

A comunicação entre os principais componentes ocorre por meio da rede, permitindo, assim, a possibilidade de um desenvolvimento distribuído, além de facilitar a utilização de contêineres, como os do Docker[1]. Outro critério levado em consideração na elaboração da arquitetura foi a possibilidade de extensão de suas funcionalidades, permitindo que ferramentas de terceiros operem sobre as informações da aplicação a ser desenvolvida.

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

4.4 Escolha das Tecnologias Envolvidas na Solução e sua Implementação

Os aspectos mais relevantes considerados no processo de seleção das tecnologias e ferramentas empregadas na implementação proposta neste trabalho foram:

1. Serem de código aberto e mantidas na data de elaboração da presente dissertação;
2. Bem consolidadas e com comunidade de usuários ativa;
3. Que evidenciem a capacidade da arquitetura de adaptar-se as seus diferentes contextos e processos internos;
4. De fácil interoperabilidade e integração, principalmente nos componentes próprios à solução e que não ficam a cargo da escolha do usuário; e
5. Que apliquem os conceitos apresentados de forma clara e cujos propósitos sejam bem definidos.

Em relação à definição de um MS, optou-se, como proposta inicial, pelas principais configurações utilizadas nos contextos menos incomuns, evidenciando característica de padrões já conhecidos. Uma importante seção dessa definição é a interface de comunicação, pois, como descrito em Sommerville (2011), ao tratar do projeto de interface de um serviço os dois primeiros passos são o projeto lógico da interface e o projeto das mensagens. O primeiro envolvendo a identificação das operações associadas ao serviços com suas entradas e saídas e o segundo especificando a estrutura de todas as mensagens utilizadas na comunicação. Ambos, ao estarem presente na definição do MS, permitem retratar bem o serviço o que facilita a compreensão e manutenção.

4.5 Elaboração e Implementação dos Casos de Uso

A partir da solução elaborada, optou-se por um caso de uso simples e de fácil entendimento, mas que ilustrasse bem as características gerais da solução para uma posterior avaliação. Tais características podem ser resumidas a:

- Uso de diferentes tecnologia e linguagens nos serviços;
- Comunicação entre serviços escritos em diferentes linguagens que possuem diferentes tipos de entidades;
- Integração com diferentes sistemas de bancos de dados;
- Implementação de processos relacionados ao *gateway*, como a autenticação.

5 FRAMEWORK SERVICER

A solução aqui proposta foi denominada Servicer e nas seguintes seções serão apresentados seus requisitos, arquitetura e implementação, respectivamente. O requisitos foram elencados como explicado na seção 4.2 e, com base neles, foi modelada uma arquitetura geral visando ser aplicada com diferentes tecnologias e diferentes protocolos e modelos de comunicação. Com base nessa arquitetura, foi escolhido um conjunto de ferramentas para implementá-la e construir uma plataforma funcional.

5.1 Requisitos

O requisitos do Servicer foram divididos em funcionais e não funcionais, uma classificação comum em requisitos de *software* (SOMMERVILLE, 2011).

5.1.1 Requisitos Funcionais

RF.1 O sistema deve compartilhar modelos de entidades entre os microsserviços de diferentes linguagens.

RF.2 O usuário deve ser capaz de definir os microsserviços nos diferentes *frameworks* utilizando o conjunto comum de configurações em alto nível.

RF.3 O usuário deve ser capaz desenvolver microsserviços em diferentes linguagens e *frameworks* considerando uma comunicação transparente entre eles.

RF.4 O sistema deve possibilitar a instalação de extensões de terceiros para resolver problemas específicos.

RF.5 O usuário deve reaproveitar o conhecimento já existente acerca dos *frameworks* suportados.

RF.6 O sistema deve disponibilizar o status da disponibilidade de cada serviço.

RF.7 O usuário deve definir em alto nível os pontos de acesso externo aos serviços, permitindo configurações de autenticação, regras de roteamento e redirecionamento, balanceamento de carga, entre outras.

5.1.2 Requisitos Não Funcionais

Cada requisito funcional acima foi mapeado para um conjunto de requisitos não funcionais, como segue.

RF.1:

– Deve haver um formato comum e independente de linguagem para especificação dos modelos de entidades de cada MS.
– Cada projeto deve converter os seus respectivos modelos definidos na linguagem de sua implementação para o formato da especificação e enviá-los à plataforma.
– A plataforma deve manter o registro de todos os modelos utilizados nos diversos microsserviços.

RF.2:

– Deve haver um conjunto de configurações gerais para a definição de um MS.
– Cada projeto é responsável por converter a especificação do MS feita pelo usuário nas estruturas do *framework* utilizado e por enviar essa definição à plataforma.
– A plataforma deve manter o registro das definições de todos os microsserviços da aplicação.

RF.3:

– Cada projeto deve conhecer os serviços externos dos quais os seus microsserviços dependem e convertê-los em serviços internos.
– A plataforma deve manter o registro dos endereços de cada MS para realizar a descoberta dos serviços.

RF.4:

– Plataforma deve compartilhar uma interface provendo as informações que possui acerca da aplicação, para que extensões criadas por terceiros possa implementar novas funcionalidades ao consumir os dados dessa interface.

RF.5:

– Todas as principais estruturas de dados e APIs dos *frameworks* utilizados devem ser estar disponíveis para o usuário durante a implementação do serviço. Apenas os processos de definição da API e das configurações do MS devem ser abstraídos.

RF.6:

– Cada MS deve implementar uma estratégia de *health check* e enviar o status da sua disponibilidade para a plataforma.
– A plataforma deve manter o registro das disponibilidades de todos os MS.

RF.7:

– A plataforma deve disponibilizar um serviço que implementa uma estratégia de *API Gateway*, permitindo a configuração de rotas, *middlewares*, regras de mapeamento das

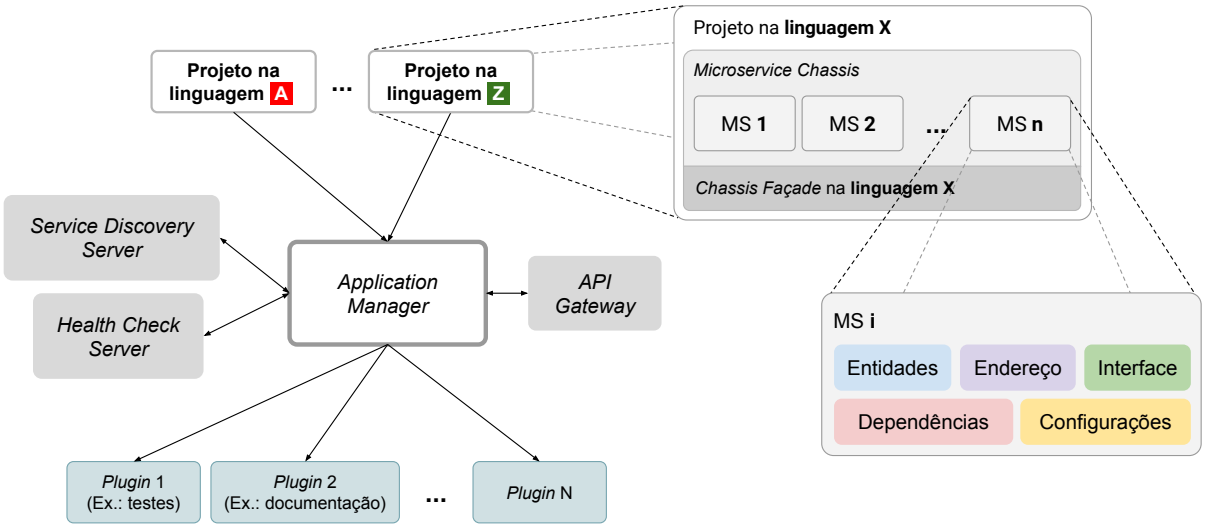
requisições para os serviços, entre outras, sem a necessidade de programação em um linguagem específica.

5.2 Arquitetura Geral

Neste trabalho é proposto uma arquitetura para possibilitar um desenvolvimento mais rápido de microsserviços construídos em diferentes linguagens de programação e seus respectivos *microservice chassis*. Como é possível observar na Figura 3, tal arquitetura é composta por:

- Um conjunto de projetos;
- Um componente central chamado *Application Manager* responsável por gerenciar a aplicação e seus módulos;
- Um componente responsável por implementar a descoberta de serviço das diferentes linguagens, chamado *Service Discovery Server*;
- Um componente responsável por implementar a checagem de disponibilidade dos serviços, chamado *Health Check Server*;
- Um componente responsável por implementar o *API Gateway* da aplicação gerada e
- Um conjunto de *plugins*.

Figura 3 – Arquitetura geral do Servicer.



Fonte: Elaborado pelo autor

Um projeto contém a implementação do seu respectivo MSC, a qual inclui o conjunto de microsserviços pertencentes ao projeto em questão e o módulo *Chassis Facade* criado na linguagem em questão.

Na visão do *Application Manager*, cada microsserviço é definido por:

- um endereço de rede (*IP* e porta);
- um conjunto de entidades, em que uma entidade representa a estrutura do dado a ser transmitido nas mensagens do MS;
- uma interface de comunicação, podendo especificar uma interação síncrona ou assíncrona, com mensagens textuais ou binárias,
- um serviço que informe o status da disponibilidade do MS e
- suas dependências: a lista de outros microsserviços dos quais ele depende, podendo estes estarem presentes em qualquer projeto dentro da atual aplicação.

5.2.1 Responsabilidades de Cada Componente

Cada componente apresentado anteriormente tem um objetivo especifico, gerencia determinadas informações e provê certas funcionalidades, como descrito a seguir.

5.2.1.1 Microservice Chassis

Aqui, entende-se MSC como descrito na subseção 2.2.3 e suas responsabilidades são as mesmas das descritas em Richardson (2019). Alguns exemplos principais são descoberta dos serviços, *Circuit Break*, *logging*, *Health Check*, entre outros. Além da implementação desses padrões específicos do contexto de microsserviços, existem as funções características de um servidor, seja ele HTTP ou não, como o tratamento das requisições e o seu repasse para os métodos dos serviços responsáveis por implementar a lógica de negócio.

5.2.1.2 Chassis Façade

O *Chassis Façade* tem duas funções principais. A primeira é a tradução da definição em alto nível do MS realizada pelo usuário para API do MSC, permitindo que, independente da linguagem a ser utilizada, o MS será definido seguindo o mesmo conjunto de configurações. Essa estratégia é a mesma presente no padrão de projeto *Façade* (SOMMERVILLE, 2011), o qual inspirou o nome do componente. A segunda função é o envio das definições dos microsserviços ao *Application Manager*, sempre que atualizada pelo desenvolvedor.

5.2.1.3 Application Manager

Em posse das definições de todos os microsserviços presente na aplicação, o *Application Manager* executa algumas tarefas com base nas informações a ele providas, tais como:

1. a partir dos **endereços**, realizar a descoberta dos serviços;
2. a partir dos serviços de checagem das disponibilidade, possibilitar a verificação dos status de cada MS da aplicação;
3. já com as **interfaces**, mapear todos os métodos de comunicação de cada serviço, por exemplo, as rotas e verbos em uma comunicação REST sobre o HTTP;
4. com o conjunto de **entidades**, compartilhar os modelos de dados entre os microsserviços e assim, juntamente com a interface, são criados automaticamente os clientes dos microsserviços externos;
5. e, com a lista de **dependências**, identificar previamente quais serviços são dependentes de serviços externos e assim os gerar os clientes sob demanda.

Em um ambiente de **desenvolvimento**, todas as tarefas descritas acima são realizadas, porém, em **produção**, apenas as tarefas 1 e 2 estão disponíveis.

5.2.1.4 API Gateway

Este componente deve implementar o controle de acesso aos serviços da aplicação final utilizando-se do padrão de mesmo nome, como descrito por Richardson (2019). É de sua responsabilidade o roteamento, processamento e tratamento das requisições, entre outras funções. Seus usos principais são para adição de autenticação, autorização, *requests logging*, *caching*, etc.

5.2.1.5 Service Discovery Server

Este servidor é responsável por manter o registro da localização de todos os microsserviços na rede. No caso, a estratégia seguida aqui é a descoberta dos serviços do lado do cliente (RICHARDSON, 2019).

5.2.1.6 Health Check Server

É um servidor cujo propósito é permitir a verificação da disponibilidade de cada serviço. Como também descrito em Richardson (2019), *Health Check* é um padrão que permite saber se um determinado serviço em execução está apto a receber requisições. Essa informação

é provida por uma checagem, implementada dentro do próprio serviço, da disponibilidade dos recursos principais que o MS depende, sejam outros serviços, bancos de dados, etc.

5.2.1.7 Plugins

Por fim, o *plugin* é o componente capaz de adicionar novas funcionalidades à plataforma. Como o *Application Manager* possui uma visão geral da aplicação e dos diversos microsserviços, este é capaz de compartilhar essas informações para módulos especializados em determinadas tarefas como geração de testes de estresse, documentação da API dos microsserviços, visualização do grafo de comunicação, entre outras.

5.2.2 Fluxo dos Dados

Para exemplificar o fluxo dos dados durante o desenvolvimento, é apresentado o cenário descrito na Figura 4. Nele, foi criado o microsserviço *X* no projeto *A*, que possui em suas dependências o microsserviço *Y* de outro projeto. O *Chassis Façade* de *A* coleta as informações acerca do MS *X* e as envia ao *Application Manager*, o qual registrará o seu endereço no *Service Discovery Server*, informará ao *Health Check Server* como verificar sua disponibilidade e buscará os dados da interface de comunicação e entidades do MS *Y*, já que ele está nas dependências de *X*. Após isso, essas informações são enviadas ao *Chassis Façade* de *A*, para que ele possa criar, na sua linguagem e *frameworks*, o cliente para o MS *Y* e, assim, proporcionar uma comunicação transparente.

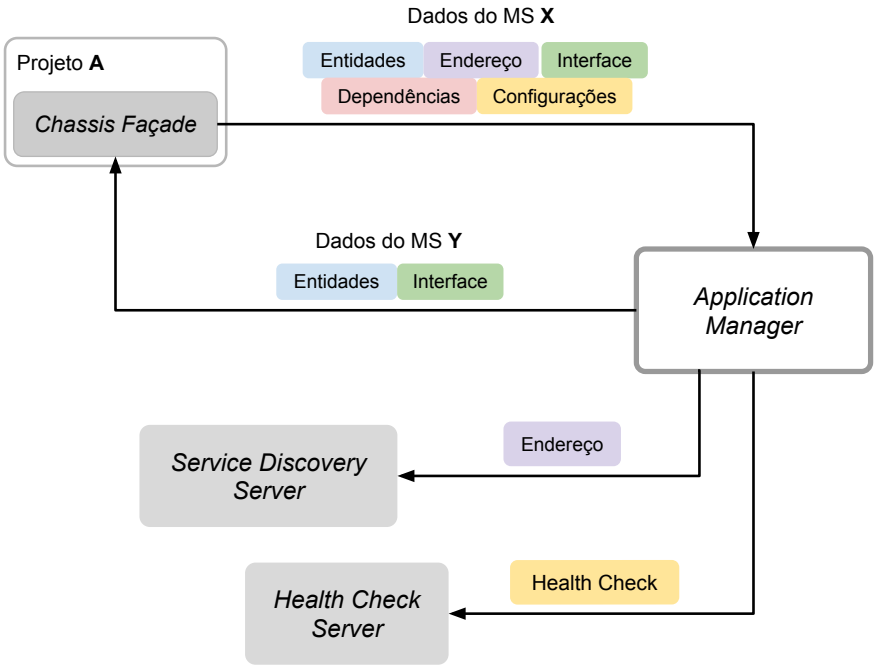
5.3 Implementação

As seguintes seções apresentam em detalhes, respectivamente, as tecnologias aplicadas no *framework* Servicer e as implementações do *Application Manager*, dos *Chassis Façades*, do *API Gateway* e de um exemplo de *Plugin*.

5.3.1 Definição de um MS

A definição de um MS contém um conjunto de configurações majoritariamente comum entre as diferentes tecnologias utilizadas nas implementações. Algumas variações podem estar presentes quando as configurações de um determinado *framework* não se enquadram na de-

Figura 4 – Fluxo das informações durante o desenvolvimento.



Fonte: Elaborado pelo autor

finição geral. Como é apresentado no Código-fonte 1, a definição é realizada no formato YAML[1], um dos formatos mais utilizados para configurações. Sua escolha deu-se pela simplicidade de leitura e compreensão, além de já ser amplamente utilizado em vários contextos. No entanto, também seria possível criar uma DSL para esta definição, substituindo, assim, apenas a camada de *parsing* e permitindo uma sintaxe mais rica.

A configuração de um serviço é composta por:

- nome do serviço (*'name'*), utilizado como identificador em toda a aplicação;
- um conjunto de entidades (*'entities'*), em que cada entidade é definida pelo seu nome e pela sua definição, a qual pode variar de acordo com a linguagem. Em Scala, por exemplo, a definição é a classe que implementa o modelo daquela entidade;
- um conjunto de rotas HTTP (*'routes'*), em que cada rota é definida pelo método do HTTP, o *path* da requisição, a função responsável processar e responder a requisição, a entidade presente no corpo da requisição e a entidade no corpo da resposta. Em relação à função, esta é referenciada de forma diferente em cada linguagem, por exemplo, em Scala é passado a classe juntamente com o método que responderá à requisição. Já em relação às duas entidades, a notação utilizada é o nome da entidade definido previamente, podendo ser envolta por colchetes, denotando um lista;

[1] https://yaml.org/

- *endpoint* para checagem de disponibilidade (*'healthCheck'*), representado pelo seu *path*, assumindo a utilização do método GET;
- configurações de roteamento do *gateway* (*'gatewayRouting'*), em que as configuração aqui possíveis são as mesmas possíveis na definição de um *Router*[2] no Traefik. O Traefik é apresentado a seguir na subseção 5.3.2.2 e a forma como ele é configurado no Servicer é descrito na subseção 5.3.3.4.
- uma lista de dependências, em que cada item da lista é o nome de outro serviço do qual este depende.

Código-fonte 1 – Exemplo de definição de um microsserviço no Servicer.

```
service:
  name: books-info-service
  entities:
    - name: Book
      definition: br.com.library.booksinfo.api.models.Book
    - name: BookForm
      definition: br.com.library.booksinfo.api.models.BookForm
  routes:
    - httpMethod: GET
      path: /api/catalog
      handlerId: br.com.library.booksinfo.api.BooksInfoServiceApi.catalog()
      requestEntity: none
      responseEntity: [Book]
    - httpMethod: GET
      path: /api/catalog/:bookId
      handlerId: br.com.library.booksinfo.api.BooksInfoServiceApi.bookDetails(id: bookId)
      requestEntity: none
      responseEntity: Book
      circuitBreaker: default
  healthCheck: /health
  gatewayRouting:
    rule: PathPrefix("/livros")
    middlewares: books-stripprefix
  dependencies:
    - books-rating-service
forReuse:
  circuitBreaker:
    default:
      enabled: true
      maxFailures: 10
      callTimeout: 10
      resetTimeout: 15
  gateway:
```

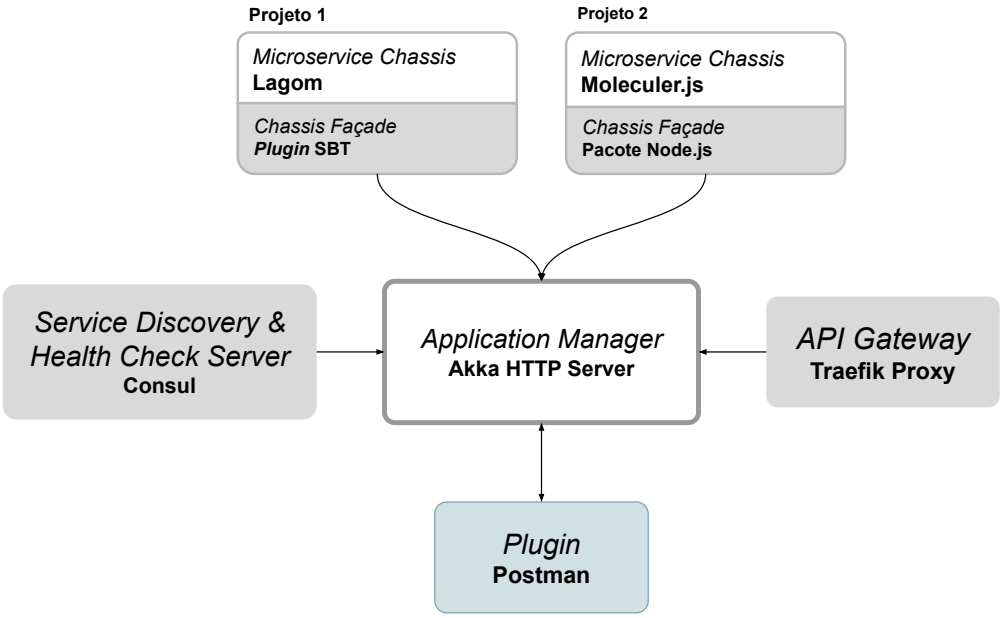
[2] https://doc.traefik.io/traefik/routing/routers/

```
34      middlewares:
35        books-stripprefix:
36          stripprefix:
37            prefixes: /livros
```

5.3.2 *Divisão da Responsabilidades*

Para cada um dos componentes apresentados na seção anterior foram escolhidas determinadas tecnologias. O paralelo com a arquitetura geral é apresentado na Figura 5. Essas tecnologias implementam a responsabilidade do seu respectivo componente e, ao integrarem-se, compõem a plataforma Servicer apresentada neste trabalho. A seguir, a lista de ferramentas utilizadas.

Figura 5 – Implementação da arquitetura geral.



Fonte: Elaborado pelo autor

5.3.2.1 *Service Discovery e Health Check*

Na busca por implementações desses dois padrões, encontrou-se o Consul[3], uma ferramenta que, além de outras funcionalidades, provê a descoberta de serviços e o monitoramento de suas disponibilidades. Logo, seu papel é implementar as responsabilidades tanto do *Service*

[3] https://www.consul.io/

Discovery Server quanto do *Health Check Server*. Além do fato de prover ambas as funcionalidades citadas, os motivos de sua escolha foram: ser código aberto, estar bem estabelecido na comunidade provendo integrações com diversas outras ferramentas, possuir um *dashboard* para visualização dos serviços e seus respectivos status, permitir a adição de *tags* e metadados aos serviços e ser de fácil utilização. Além disso, em comparação com o Eureka[4], outra ferramenta bastante usada nesse contexto, o Consul apresenta uma maior taxa de transferência e menor tempo de resposta, como é apontado por Al-Debagy e Martinek (2018).

5.3.2.2 API Gateway

Diante das várias possibilidades de tecnologias que realizam esse papel, optou-se pelo Traefik Proxy[5] ou simplesmente Traefik, que é uma solução de código aberto desenvolvida com o objetivo de facilitar o direcionamento de requisições para os componentes responsáveis por respondê-las (Hyun *et al.*, 2018). Com o Traefik é possível criar regras de redirecionamento e *middlewares* apenas por meio de configuração, sem a necessidade de programar em uma linguagem específica, tornando-o uma ótimo opção para a estratégia de múltiplas linguagens. Além disso, é possível definir os serviços e suas respectivas configurações por intermédio do Consul, fazendo com que essa integração seja natural.

5.3.2.3 Microservice Chassis

Em relação às implementações dos serviços, optou-se em realizá-las nas linguagens Javascript e Scala. Enquanto Scala é uma linguagem compilada, fortemente tipada e que executa sobre a *Java Virtual Machine* (JVM), Javascript é interpretada e fracamente tipada. Essa diferença é destacada com o intuito de evidenciar a capacidade de integração do Servicer. Dois *frameworks* conhecidos nessas linguagens são o Lagom e o Moleculer.js.

Disponível em Java e Scala, o Lagom Framework provê uma coleção de ferramentas que juntas compões um ambiente robusto para o desenvolvimento de microsserviços. Neste trabalho, optou-se pela sua versão em Scala para a utilização como *Microservice Chassis*. O Lagom traz soluções próprias para os diversos problemas relacionados a microsserviços, dificultando assim a integração com outras ferramentas de terceiros. Por conta disso, ele foi considerado uma ótima opção para demonstrar a flexibilidade da plataforma, ao integrá-lo com

[4] https://github.com/Netflix/eureka
[5] https://doc.traefik.io/traefik/

os demais componentes da arquitetura proposta.

Já o Moleculer.js[6] é um *framework* para criação de microsserviços em Javascript com o Node.js. Assim como o Lagom, ele traz um ambiente completo e robusto (Bigheti *et al.*, 2019) e, por isso, também desempenhará o papel de *Microservice Chassis*. No entanto, sua escolha se deu principalmente pelas diferenças entre o Lagom.

5.3.2.4 Chassis Façades

Cada umas das linguagens escolhidas têm suas próprias ferramentas para gerenciamento de dependências e execução, por isso, os *Chassis Façade* foram desenvolvidos em duas ferramentas consideradas as mais adequadas no contexto de cada linguagem.

A primeira delas é o Node.js[7], uma plataforma para execução de código Javascript fora do contexto de navegadores, tanto no ambiente de desenvolvimento quanto no de produção. Um dos seus principais usos tem sido para a criação de aplicações *web* (Tilkov; Vinoski, 2010). Utilizando o Node.js, foi desenvolvido um pacote usado como dependência do projeto em que o Moleculer.js estiver presente. A sua instalação está disponível através do *Node Package Manager* (NPM)[8].

Já para o Lagom, foi utilizado o *Scala Build Tool* (SBT)[9], uma ferramenta que auxilia no gerenciamento das dependências, *plugins* e tarefas de projetos de *software* escrito em Scala ou Java. Os projetos Lagom são criados sobre o SBT e, por isso, ele é utilizado para a criação de um *plugin* que faz o papel de *Chassis Façade*.

Cada *Chassis Façade* implementa cinco funções principais:

CF.1 Registro das entidades: conversão das entidades em JSON Schema e envio ao *Application Manager*.

CF.2 Registro da definição do MS: em que são enviadas todas as informações descritas na definição do MS, envolvendo rotas e demais configurações.

CF.3 Integração à descoberta de serviços: integrar-se a estratégia de descoberta de serviços utilizada pelo MSC e atualizar o catálogo de serviços geral enviando os dados da localização ao *Application Manager*.

CF.4 Geração do cliente de um MS externo: essa função é responsável por criar o cliente do

[6] https://moleculer.services/
[7] https://nodejs.org/
[8] https://npmjs.com/
[9] https://www.scala-sbt.org/

microsserviço externo ao presente no seu *framework*.

CF.5 Conversão da definição comum nas estruturas do MSC: ler a definição e traduzir utilizando as funções disponíveis em cada *framework*.

5.3.2.5 Exemplo de Plugin

O exemplo de *plugin* desenvolvido utiliza o Postman[10],uma das principais ferramentas utilizadas durante o desenvolvimento de APIs REST, capaz de auxiliar durante as etapas de testes, construção e refatoração dessas APIs (RANGA; SONI, 2019). Ele conta com um ambiente para a execução de requisições onde é possível criar coleções de serviços e organizá-las em diferentes diretórios. Com essas coleções, também é possível criar documentações sobre a API e, utilizando o pacote Newman[11], elaborar testes automatizados. O atual *plugin* possibilita a criação automática de coleções a partir das definições dos microsserviços.

5.3.2.6 Servidor Application Manager

O papel do *Application Manager* é desempenhando por um servidor criado em Akka HTTP[12]. Ele é responsável pela comunicação com os *plugins* e os *Chassis Façades* de cada projeto, a qual é feita por meio uma API REST. A sua escolha se deu pela sua flexibilidade, simplicidade e robustez. Por não se tratar de um *framework web* e sim de um *toolkit*, é possível utilizá-lo apenas para construir camadas de integração com base no HTTP, sem que a modelagem da aplicação seja enrijecida por alguma exigência arquitetural.

5.3.2.7 Gerenciamento das Entidades

A atual implementação assume que a comunicação entre os serviços dá-se sobre o HTTP e com mensagens no formato JSON. Logo, para que o esquema dos dados possa ser gerenciado pelo *Application Manager* e para que ele possa compartilhá-los entre cada MS, adotou-se a especificação JSON Schema[13]. Com essa especificação é possível representar o esquema das entidades no formato JSON, que armazenará os campos, tipos, formatos e restrições dos dados (RICHARDSON, 2019). Além disso, como essa comunicação entre os projetos e o *Application Manager* ocorre pela rede, as entidades já estarão serializadas para a transferência.

[10] https://www.postman.com/
[11] https://github.com/postmanlabs/newman
[12] https://doc.akka.io/docs/akka-http
[13] http://json-schema.org/

5.3.3 Componentes Implementados

A seguir, os detalhes da implementação proposta de cada componente da arquitetura.

5.3.3.1 Akka HTTP Server

Como primeira responsabilidade do *Application Manager*, tem-se o gerenciamento das informações dos serviços e entidades, como descrito na subseção 5.2.1.3. Para isso, foi criado um servidor HTTP que provê uma API REST consumida tanto pelos *Chassis Façades* quanto pelos *Plugins*. A sua lista de *endpoints* está descrita na Figura 6. Com essa API é possível ter acesso às definições de todos os microsserviços no formato apresentado na subseção 5.3.1 e, além disso, é possível também atualizar constantemente essas definições durante o desenvolvimento.

Figura 6 – *Endpoints* da API Rest do *Application Manager*.

```
POST   /services - Registra um novo serviço.
GET    /services - Retorna a lista de serviços registrados.
GET    /services/:serviceId - Detalha um serviço.
PUT    /services/:serviceId - Atualiza os dados de um serviço.
DELETE /services/:serviceId - Remove um serviço.
PUT    /services/:serviceId/discovery - Atualiza o endereços do serviço.
GET    /dev/entities - Retorna a lista de entidades cadastradas.
POST   /dev/entities - Registra uma nova entidade.
GET    /dev/entities/:entityId - Detalha uma entidade.
PUT    /dev/entities/:entityId - Atualiza os dados de uma entidade.
DELETE /dev/entities/:entityId - Remove uma entidade.
```

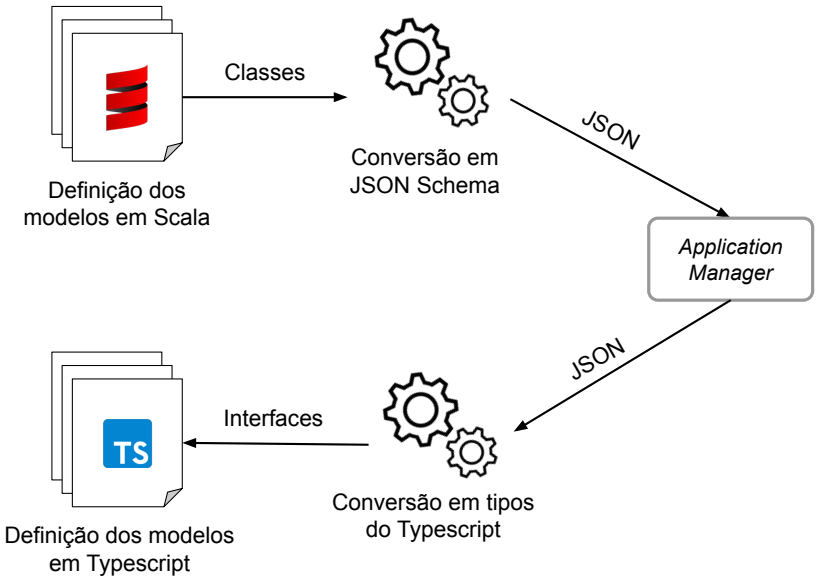
Fonte: Elaborado pelo autor

Para realizar a descoberta dos serviços, o servidor comunica-se com a API do Consul por meio de um *Software Development Kit* (SDK) e registra cada MS em seu catálogo. Além disso, é configurado também o *Health Check* a partir da informação provida na definição do MS. Com isso, o Consul é capaz de realizar a checagem periódica da disponibilidade dos serviços. A comunicação do *Application Manager* com o Consul também provê a configuração do Traefik, como é descrito na subseção 5.3.3.4.

Já as entidades são armazenadas em memória no formato JSON seguindo a especificação JSON Schema, como explicado na subseção 5.3.2.7. Assim, os *Chassis Façades* requisitam os esquemas das entidades dos microsserviços externos à sua plataforma, como descrito no fluxo apresentado na subseção 5.2.2, e geram os modelos na sua linguagem. Esse fluxo é exemplificado na Figura 7, onde ilustra-se o caso em que é um MS escrito em Typescript

com o Moleculer, depende de outro escrito em Scala com o Lagom. Vale ressaltar que tanto as operações de conversão da especificação em JSON Schema para a linguagem alvo quanto o processo inverso são responsabilidades dos *Chassis Façades*, descritos a seguir.

Figura 7 – Fluxo da geração automática de entidades compartilhadas entre os serviços do Servicer.



Fonte: Elaborado pelo autor

5.3.3.2 SBT Plugin

A seguir, é descrito como as funções apresentadas na subseção 5.3.2.4 foram implementadas no *plugin*. Primeiramente, devido à necessidade de comunicar-se com o *Application Manager*, foi desenvolvido um SDK para gerenciar as requisições HTTP. O SDK traduz os *endpoints* descritos na Figura 6 em métodos executáveis.

O SBT possui o conceito de Grafo de Tarefas (Task Graph[14]) em que as tarefas, como compilação (*'compile'*) e remoção dos artefatos gerados (*'clean'*), são encadeadas em uma sequência que representam a ordem das suas execuções de forma síncrona. Desse modo, é possível criar tarefas customizadas e inseri-las nesse grafo permitindo que elas ocorram antes ou depois de outras. Essa estratégia foi utilizada no desenvolvimento do *plugin* em que foram criadas uma tarefa para cada um das funções **CF.1**, **CF.2**, **CF.4** e **CF.5**. Todas essas tarefas são executadas antes da tarefa nativa *'Compile / sourceGenerators'*, a qual possibilita a geração de código-fonte inserido no processo de compilação e que não é gerenciado pelo usuário. A

[14] https://www.scala-sbt.org/1.x/docs/Task-Graph.html

execução da função **CF.3** é descrita a seguir.

Em relação a função **CF.1**, foi utilizado o módulo de terceiros "scala-jsonschema"[15], o qual lê as classes escritas em Scala e converte para a especificação JSON Schema. Assim, sempre que o projeto for recompilado, qualquer alteração na definição dos modelos (no caso do Scala, expresso em classes) será automaticamente enviada ao *Application Manager*, garantindo assim que ele sempre possua a versão mais atual das entidades.

Já na função **CF.2**, a cada compilação a definição escrita em YAML é convertida em JSON e, em seguida, enviada ao *Application Manager* utilizando o SDK desenvolvido.

Para a execução da função **CF.4**, primeiramente são verificados quais dos serviços presentes na lista de dependências não pertencem ao projeto atual. Em seguida, é requisitado ao *Application Manager* tanto a definição desses serviços quanto a especificação das entidades utilizadas nos seus *endpoints* HTTP. Em posse desses dados, o *plugin* converte o JSON Schema das entidades para classes e gera alguns arquivos para permitir que tanto a definição dos serviços externos utilizando as estruturas do Lagom quanto os modelos estejam presentes no *classpath* da aplicação.

Vale lembrar que apesar de haver geração de código, nenhum desses artefatos é gerenciado pelo desenvolvedor e que todos esses processos ocorrem fora do código principal da aplicação. Essa estratégia é a mesma utilizada no *framework* Play[16] no qual o Lagom se baseia. Dessa forma, é possível inserir nas lógicas dos serviços escritos no Lagom chamadas aos serviços escritos, por exemplo, no Moleculer.

Por fim, a função **CF.5** foi implementada também gerando arquivos não gerenciados pelo desenvolvedor, mas que garantem a execução dos serviços escritos em YAML da mesma forma que os descritos programaticamente, em um projeto tradicional. A única diferença é que foi adicionado o processo de registro do serviço no *Application Manager* quando a aplicação é iniciada, para a descoberta dos serviços. Logo, é dessa forma que é realizada a função **CF.3**.

5.3.3.3 Módulo Node.js

Como é exposto a seguir, a implementação das funções da subseção 5.3.2.4 possui algumas diferenças no módulo Node.js. Apesar disso, há uma semelhança: o desenvolvimento de um SDK com o mesmo propósito ao citado na seção anterior, porém escrito em Typescript.

[15] https://github.com/andyglow/scala-jsonschema
[16] https://www.playframework.com/

Para a função **CF.1**, foi também utilizado um módulo de terceiros, no caso o "typescript-json-schema"[17], o qual converte as interfaces e tipos escritos em Typescript para a especificação JSON Schema. Assim, sempre que houver alterações nos arquivos de definições dos modelos, essas diferenças serão persistidas automaticamente no catálogo de entidades no *Application Manager*. A possibilidade de observar por alterações nos arquivos é provida pelo próprio ambiente de desenvolvimento do Node.js juntamente com o do Typescript.

Em relação à função **CF.2**, o mesmo ocorre com a definição escrita em YAML, que quando modificada é convertida em JSON e, em seguida, enviada ao *Application Manager* utilizando o SDK desenvolvido em Typescript.

Já na função **CF.4**, o fluxo é semelhante ao que ocorre no *plugin* do SBT: são verificados os serviços dos quais o atual depende e que não pertencem ao projeto atual. Depois, é requisitado ao *Application Manager* tanto a definição desses serviços quanto a especificação das entidades utilizadas nos seus *endpoints* HTTP. Em posse desses dados, o *plugin* converte o JSON Schema das entidades para tipos do Typescript, porém, no caso da definição do MS esta é criada completamente em memória, não havendo a necessidade de geração de arquivo.

A função **CF.5** foi implementada lendo a definição YAML e convertendo em memória para as estruturas de dados do Molecular. Logo, não há um passo intermediário na compilação nesse processo, tudo ocorre em tempo de execução. Essa diferença realça bem a independência dos *Chassis Façades* uns aos outros, pois seus objetivos podem ser concretizadas da melhor maneira para cada tecnologia.

Por fim, a função **CF.3** foi realizada implementando cada método da Interface do Molecular para *Service Discovery* com o Consul, comunicando-se com sua API.

5.3.3.4 Traefik - API Gateway

Diferente de outras ferramentas similares, o Traefik permite que a configuração de roteamentos, *middlewares* e serviços seja feita de maneira dinâmica, ou seja, o processo do Traefik não precisa parar para que mudanças nas configurações ocorram. Essa dinamicidade acontece graças aos *Providers*, que informam, durante a execução, quais serviços estão presentes na infraestrutura e compartilham algumas informações sobre eles, como IP, disponibilidade, entre outras.

Os *Providers* podem ser tanto arquivos de configurações, quanto plataformas de

[17] https://github.com/YousefED/typescript-json-schema

infraestrutura como Docker e Kubernetes. Uma outra possibilidade é o próprio Consul, pois com ele é possível adicionar algumas *tags* à cada serviço da plataforma, as quais podem ser lidas pelo Traefik, permitindo que os serviços realizem suas próprias configurações. Desse modo, o *Application Manager*, em posse das configurações do *gateway* de cada MS (chave *'gatewayRouting'* da definição apresentada na subseção 5.3.1), consegue repassar essas informações ao Consul como *tags* no momento do registro. Um exemplo dessas *tags* é apresentado na Figura 8 e o fluxo completo das informações está ilustrado na Figura 9.

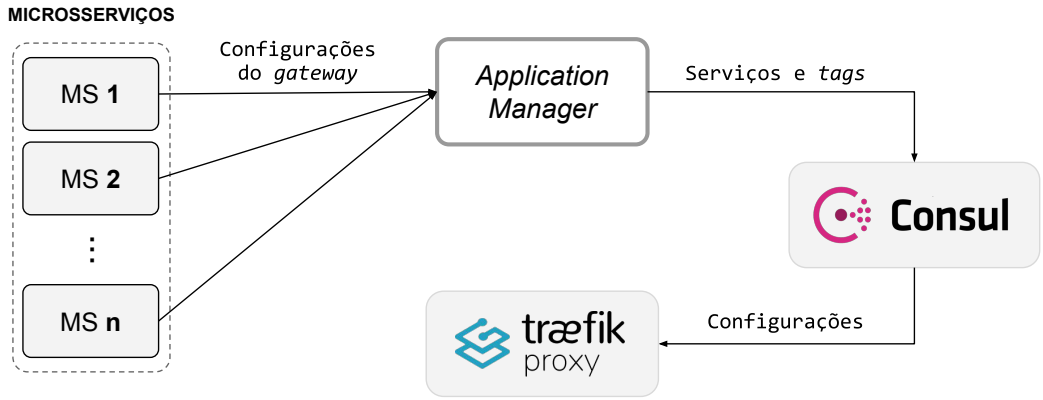
As demais configurações do Traefik que não estão atreladas à nenhum serviço, e sim utilizadas no escopo global, podem ser definidas nas configurações do *Application Manager*.

Figura 8 – Exemplo de *tags* do Consul utilizadas na configuração do Traefik.

```
- traefik.http.routers.books.rule=PathPrefix(`/livros`)
- traefik.http.routers.books.entrypoints=web
- traefik.http.routers.books.middlewares=books-stripprefix
- traefik.http.middlewares.books-stripprefix.stripprefix.prefixes=/livros
```

Fonte: Elaborado pelo autor

Figura 9 – Fluxo de configuração do Traefik.



Fonte: Elaborado pelo autor

5.3.3.5 Postman Plugin

Para este *plugin*, foi criada uma simples aplicação Node.js que requisita o *Application Manager* utilizando o SDK desenvolvido também em Node.js em busca das informações da interface dos microsserviços e a estrutura das suas entidades. Com isso, são criados *collections* do Postman contendo cada requisição e ainda detalhados, por exemplo, o modelo JSON para os *endpoints* de criação e edição de recursos. São apenas informados os serviços dos quais desejasse

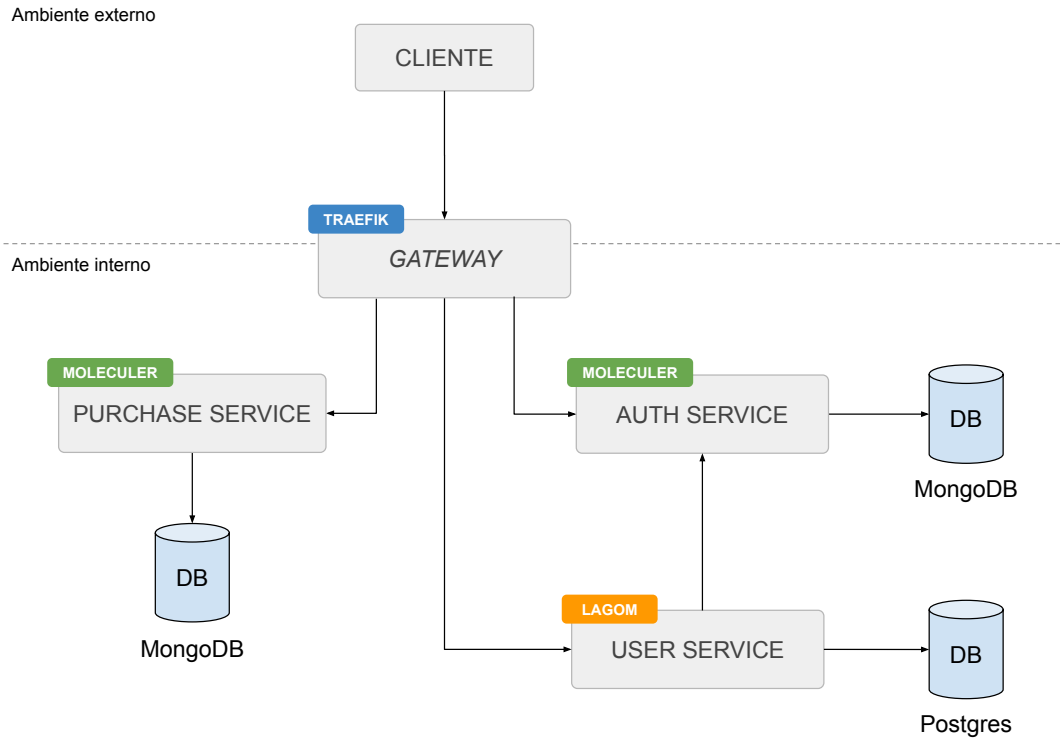
gerar e o endereço do *Application Manager* em execução, para que esses processos ocorram.

6 ESTUDO DE CASO

Como estudo de caso, foi elaborada uma aplicação orientada a microsserviços utilizando o *framework* Servicer. O intuito do estudo é avaliar se o objetivo de facilitar o desenvolvimento de microsserviços e a integração entre diferentes linguagens foi atingido, quais as limitações existentes e futuras melhorias.

Apesar de simples, a aplicação retrata um cenário comum e que evidencia as características sob os critérios avaliados. O sistema trata-se de uma lista de compras, em que um usuário autenticado (que já passou pelas etapas de criação de conta e *login*) possa adicionar futuras intenções de compra, informando uma URL para o produto em questão e uma data de previsão para a compra. A partir disso, foi elaborada a arquitetura dos serviços, apresentada na Figura 10.

Figura 10 – Diagrama dos serviços do caso de uso do Servicer.



Fonte: Elaborado pelo autor

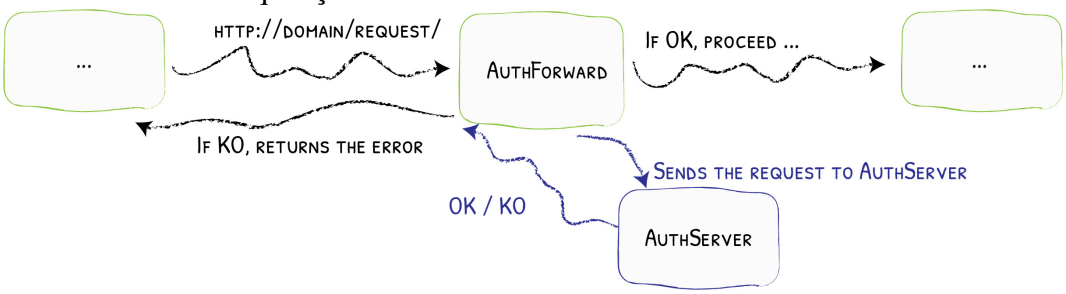
São três microsserviços:

– *'Auth Service'*: realiza o controle da autenticação dos usuários implementando uma estratégia de *token* de acesso, mais especificamente o *JSON Web Token* (JWT) (RICHARDSON, 2019). É desenvolvido no Moleculer.
– *'User Service'*: mantém os dados dos usuário e realiza a operação de criação de conta e *login*. Desenvolvido no Lagom.

– *'Purchase Service'*: mantém os dados da lista de compra do usuário e serve essas informações. Além disso, realiza algumas operações como busca dos metadados dos produtos nos sites que os ofertam. Desenvolvido no Moleculer.

O processo de autenticação, por ser uma *edge function*, ocorre no Traefik, onde foi criado um *middleware* do tipo ForwardAuth[1]. O fluxo das requisições utilizando essa estratégia é explicado na Figura 11.

Figura 11 – Fluxo das requisições no *middleware* ForwardAuth do Traefik.



Fonte: https://doc.traefik.io/traefik/middlewares/forwardauth/

Os dois serviços desenvolvidos no Moleculer utilizam o MongoDB[2] como banco de dados, enquanto o desenvolvido no Lagom faz uso do PostgreSQL[3]. A utilização de sistemas de bancos de dados diferentes objetiva exemplificar um contexto em que os serviços utilizam seus recursos de forma totalmente independente uns dos outros.

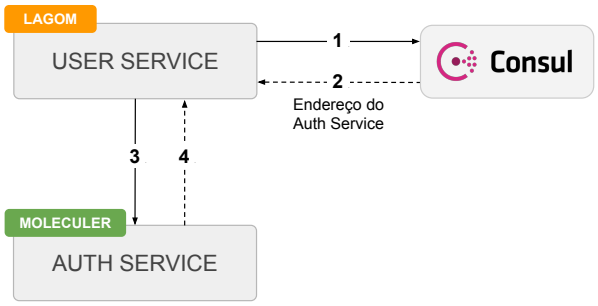
Os fluxos durante o processo de descoberta de serviços estão descritos na Figura 12, onde as respostas das requisições estão representadas pelas setas tracejadas e os número informam a ordem em que os eventos acontecem. A estratégia utilizada em ambos os casos é o *Server-side Discovery*. O primeiro caso presente na Figura 12a exemplifica a comunicação entre microsserviços, em que o *'User Service'* faz o papel de cliente enquanto o *'Auth Service'* atua como servidor. Nesse caso, o cliente precisa conhecer a localização do servidor e, por isso, a busca no catálogo do Consul o qual responderá com o endereço de alguma das instâncias do *'Auth Service'*, após aplicar um algoritmo de balanceamento de carga. Em posse da localização, o *'User Service'* agora pode fazer a requisição diretamente.

Já o caso trazido na Figura 12b exemplifica a comunicação entre um cliente externo e um MS. Nesse exemplo, a responsabilidade de buscar no Consul a localização do serviço requisitado é do *gateway*. Após receber uma requisição, o Traefik verifica nas suas configurações para qual serviço ela deve ser encaminhada (informação provida na configuração *rules* dos

[1] https://doc.traefik.io/traefik/middlewares/forwardauth/
[2] https://www.mongodb.com/
[3] https://www.postgresql.org/

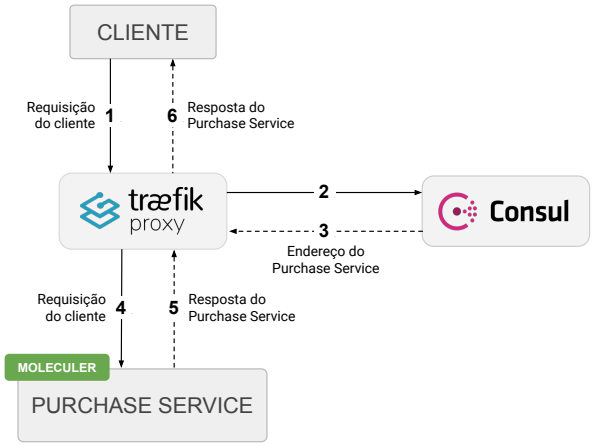
Routers). Em seguida, busca sua localização no Consul, com o qual já está integrado, como explicado na subseção 5.3.3.4. Em posse desse endereço, repassa a requisição do cliente externo para o MS, que atua como servidor. Vale ressaltar que esse fluxo considera uma requisição que não possui a restrição de autenticação.

Figura 12 – Fluxos da descoberta de serviço.



Fonte: Elaborado pelo autor

(a) Fluxo da descoberta de serviço entre micros-serviços.



Fonte: Elaborado pelo autor

(b) Fluxos da descoberta de serviço a partir do *gateway*.

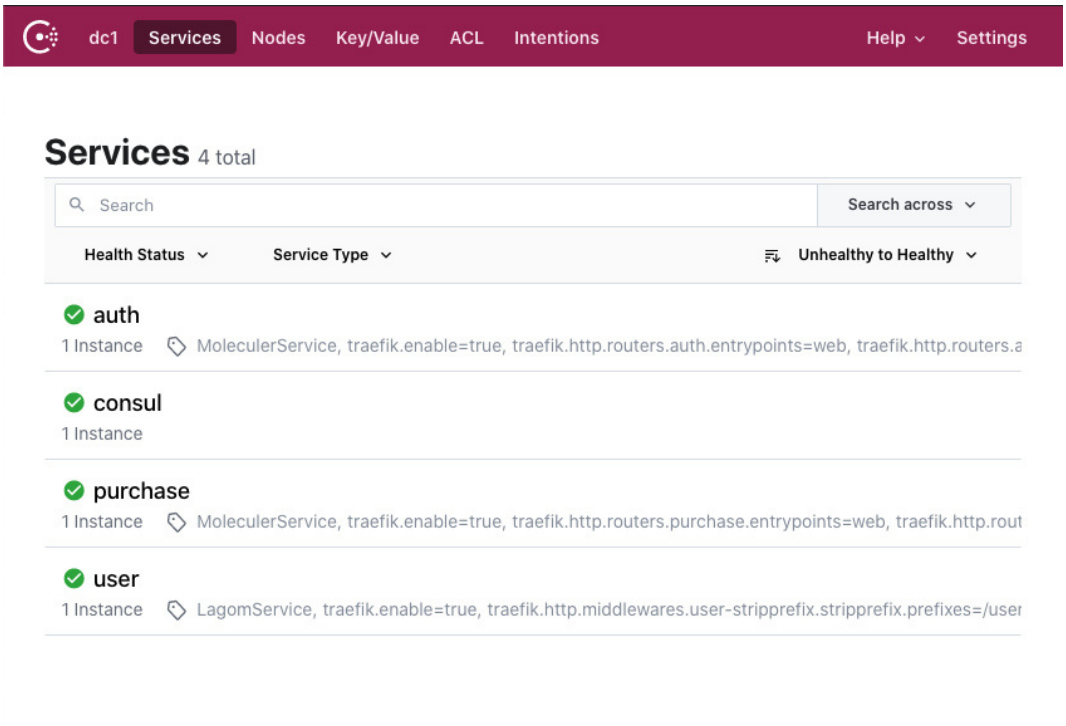
Na Figura 13 é apresentado uma captura de tela dos serviços descritos acima e registrados no Consul. Além desses, também está presente o serviço do próprio Consul, que é registrado por padrão. É possível perceber que cada MS possui uma *tag* informando em qual *framework* ele foi desenvolvido (*LagomService* e *MoleculerService*). Além dessas, há também as *tags* utilizadas para a configurações do Traefik. Outra informação disponível é o *Health Status*, em que o Consul informa o resultado da checagem de disponibilidade realizada requisitando o *endpoint* informado na configuração *healthCheck* das definições de cada serviço.

Já na Figura 14, há a captura de tela do Traefik mostrando os *Routers* configurados a partir das *tags* citadas anteriormente. É possível notar que na coluna de *Provider* as configurações não padrões possuem o logotipo do Consul, denotando a sua utilização.

Por fim, utilização do *plugin* do Postman foi de grande valia em relação ao teste durante o desenvolvimento, pois foi possível testar a comunicação e funcionalidades de cada MS separadamente sem a necessidade de reescrever as suas interfaces sempre que fossem alteradas.

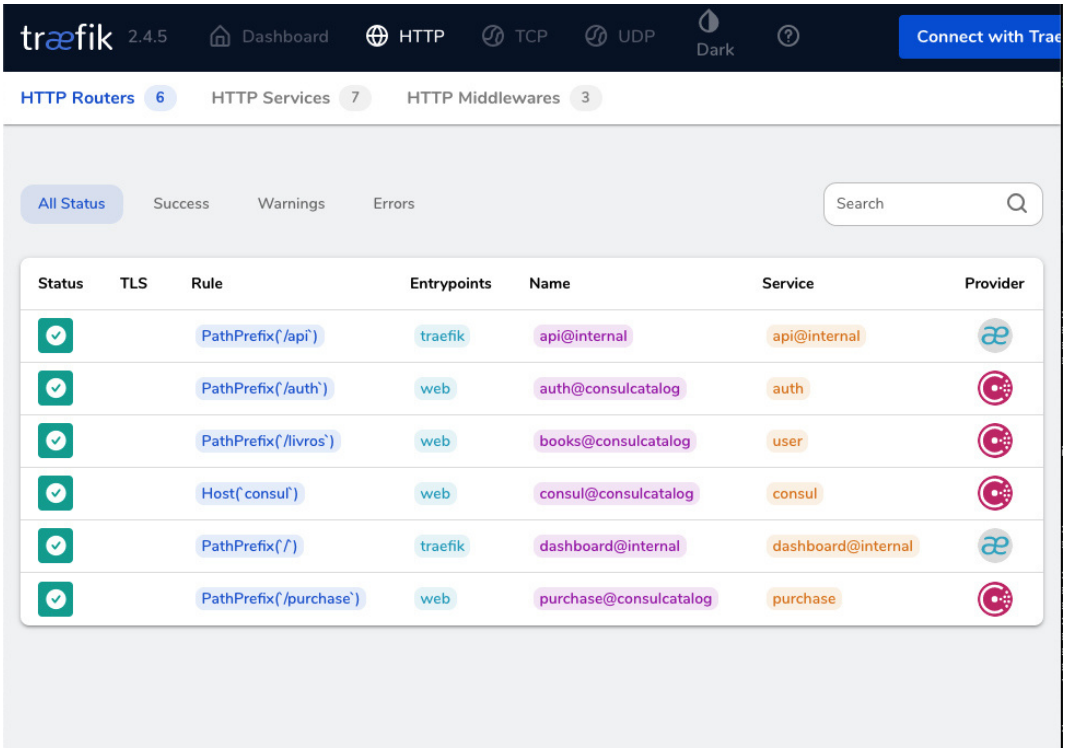
O código-fonte da aplicação exemplo e de todos os componentes do Servicer encontram-se em um repositório público, acessível pelo endereço https://github.com/samirbraga/

Figura 13 – Captura de tela do *dashboard* do Consul mostrando os serviços do caso de uso do Servicer.



Fonte: Elaborado pelo autor

Figura 14 – Captura de tela do *dashboard* do Traefik mostrando os *Routers* do caso de uso do Servicer.



Fonte: Elaborado pelo autor

servicer.

7 CONCLUSÕES E TRABALHOS FUTUROS

Com a análise da aplicação desenvolvida como estudo de caso e descrita no Capítulo 6, foi possível extrair algumas conclusões descritas a seguir.

Em relação à definição de um MS, na perspectiva do desenvolvedor, constatou-se um ganho de produtividade relacionado a esse processo. Esse resultado é percebido devido à redução de tempo na especificação das configurações do MS nos diferentes *Microservice Chassis*, pois não há a necessidade, neste momento, de conhecimento das suas estruturas e, consequentemente, também não é necessária a escrita de código na linguagem alvo.

Primeiramente, devido as configurações serem comuns a todas as linguagens e utilizarem o mesmo formato para a sua especificação, no caso YAML. Além disso, as variáveis que coordenam as principais características do MS são dadas em poucas linhas, ao invés de estarem espalhadas pelo código-fonte. Entretanto, na perspectiva da plataforma, constatou-se que a normalização dessas definições para os diferentes *frameworks* acarreta na perda de expressividade, consequência de algumas particulares de cada tecnologia.

Em suma, com a utilização do Servicer, é possível perceber que o objetivo de elaborar um conjunto de definições para microsserviços que podem ser reaproveitados em diversos contextos e ferramentas traz diversos benefícios. Desses, é possível destacar o reaproveitamento do conhecimento entre diferentes tecnologias, um enriquecimento colaborativo dessas configurações e a abstração das particularidades de implementações de cada linguagem no processo de definição do serviço.

Neste trabalho, os contextos de utilização desse conjunto de definições resumiram-se a tradução das configurações para a estrutura de dados do *framework* utilizado e ao compartilhamento dessas informações entre as diferentes linguagens para automatização de processos de integração, no entanto, demais objetivos podem ser propostos, como a criação de uma especificação formal.

Já sobre os componentes do Servicer, a existência do *Application Manager* responsável pelas configurações dos demais componentes e pelo provimento das informações gerais da aplicação é crucial. Em conjunto com os *Chassis Façades*, ambos já concretizam boa parte do objetivo de abstrair as complexidades dos processos de integração. A geração dos clientes para os serviços externos também reduzem um grande percentual do esforço no desenvolvimento da comunicação entres os serviços de diferentes linguagens.

A possibilidade de expansão das funcionalidade por meio de *plugins* também se

mostrou ser uma ideia com grande possibilidade de evolução, pois vários outros usos relacionados a análises dos dados da aplicação e operações sobres eles podem ser realizadas a fim auxiliar em tarefas de um domínio específico.

Em geral, há sempre a possibilidade de reimplementar a arquitetura geral proposta com novas tecnologias a medidas que forem surgindo, ou mesmo evoluir a arquitetura para uma melhor modularização ou para suportar outros tipos de padrões.

7.1 Trabalhos Futuros

Como trabalhos futuros planeja-se realizar os seguintes ajustes e melhorias:

- Tornar as configurações da definição de um MS relacionadas ao *gateway* mais genéricas. As atuais estão fortemente atreladas ao Traefik, não permitindo a integração de ferramentas alternativas.
- Enriquecer a definição de um MS para permitir também a especificação de configurações relacionadas à outras fases da aplicação, não somente o desenvolvimento. Por exemplo, definições relacionadas ao *deploy*, como informação sobre a imagem e o contêiner a serem utilizados; definições sobre testes ou sobre monitoramento.
- Criação de uma DSL para a definição. Apesar de YAML ser um formato bastante conhecido e de fácil utilização, acredita-se que seria vantajoso oferecer a possibilidade de utilização de uma DSL com um maior poder de expressividade, permitindo, por exemplo, o uso de variáveis e concatenações de *strings*.
- Testes unitários das implementações de cada componente.
- Criar *Chassis Façades* para outras linguagens e *frameworks*, expandindo assim as possibilidades de aplicação do Servicer.
- Permitir outros métodos de comunicação, como uma comunicação assíncrona baseada em eventos, ou ainda não baseadas em HTTP como o Protocol Buffers[1].
- Facilitar a utilização de uma comunicação indireta por meio de *message brokers*.
- Integração nativa ao *Service Mesh*. Como descrito em Richardson (2019), muitas das funcionalidades relacionadas à rede implementadas nos *Microservice Chassis* podem ser aplicadas à nível de infraestrutura como um *proxy* que intercepta as requisições dos serviços. A principal vantagem é que essas funções ficariam externas à aplicação e tratadas independente de tecnologia, o que encaixa-se perfeitamente a proposta do Servicer.

[1] https://developers.google.com/protocol-buffers

- Permitir a integração de serviços externos ao próprio Servicer. Assim, seria possível que os microsserviços gerenciados pelo Servicer se comuniquem com outros já hospedados em outras infraestruturas, os quais seriam definidos utilizando praticamente o mesmo conjunto de configurações.
- Elencar um conjunto de critérios para avaliação do Servicer em um cenário real de uso com usuários desenvolvedores de diferentes equipes.
- Avaliar uma remodelagem da arquitetura geral que considera somente a existência do *Application Manager* e de outro módulo genérico que se comunicaria com ele. Cada instância desse módulo poderia assumir tanto o papel de projeto, como de *plugin* e até dos demais componentes como o *API Gateway* e *Service Discovery Server*. Todos comunicando-se por meio de uma mesma API comum que permita a execução de todas essas funções.
- Elaborar uma interface gráfica para visualização dos componentes em desenvolvimento, com suas configurações e permitindo alterá-las.
- Avaliar a execução do Servicer também no ambiente de produção. Assim, poderiam ser realizadas análises sobre o tráfego dos dados e fluxo das informações e, a partir disso, adicionar funcionalidades com base nelas, como sugestões de melhorias na arquitetura da aplicação desenvolvida.

REFERÊNCIAS

Al-Debagy, O.; Martinek, P. A comparative review of microservices and monolithic architectures. In: **2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)**. [*S. l.*: *s. n.*], 2018. p. 000149–000154.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: an experience report on migration to a cloud-native architecture. **IEEE Software**, v. 33, 05 2016.

Bigheti, J. A.; Fernandes, M. M.; Godoy, E. P. Control as a service: A microservice approach to industry 4.0. In: **2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0 IoT)**. [*S. l.*: *s. n.*], 2019. p. 438–443.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) – University of California, Irvine, 2000.

FOWLER, M. **Richardson Maturity Model**. 2010. Disponível em: https://martinfowler.com/articles/richardsonMaturityModel.html.

FRANCESCO, P.; MALAVOLTA, I.; LAGO, P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: . [*S. l.*: *s. n.*], 2017. p. 21–30.

HALIN, A.; NUTTINCK, A.; ACHER, M.; DEVROEY, X.; PERROUIN, G.; HEYMANS, P. Yo variability! jhipster: A playground for web-apps analyses. In: **Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems**. New York, NY, USA: Association for Computing Machinery, 2017. (VAMOS '17), p. 44–51. ISBN 9781450348119. Disponível em: https://doi.org/10.1145/3023956.3023963.

HALIN, A.; NUTTINCK, A.; ACHER, M.; DEVROEY, X.; PERROUIN, G.; BAUDRY, B. Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. **Empirical Software Engineering**, v. 24, n. 2, p. 674–717, Apr 2019. Disponível em: https://doi.org/10.1007/s10664-018-9635-4.

Hyun, W.; Huh, M. Y.; Park, J. Implementation of docker-based smart greenhouse data analysis platform. In: **2018 International Conference on Information and Communication Technology Convergence (ICTC)**. [*S. l.*: *s. n.*], 2018. p. 1103–1106.

KLEPPMANN, M. **Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems**. [*S. l.*]: OReilly, 2018.

LABS, C. **Cortex - Deploy machine learning models to production**. 2021. Disponível em: https://cortex.dev/.

Liu, D.; Zhu, H.; Xu, C.; Bayley, I.; Lightfoot, D.; Green, M.; Marshall, P. Cide: An integrated development environment for microservices. In: **2016 IEEE International Conference on Services Computing (SCC)**. [*S. l.*: *s. n.*], 2016. p. 808–812.

NEWMAN, S. **Monolith to Microservices**. 1ª. ed. [*S. l.*]: O'Reilly Media, Inc., 2019.

RANGA, V.; SONI, A. Api features individualizing of web services: Rest and soap. **International Journal of Innovative Technology and Exploring Engineering**, v. 8, 08 2019.

RICHARDSON, C. **Microservices patterns: with examples in Java**. [*S. l.*]: Manning Publications, 2019.

SASIDHARAN, D. K. **Full Stack Development with JHipster - Build full stack applications and microservices with Spring Boot and modern JavaScript frameworks**. [*S. l.*]: Packt Publishing Ltd, 2020.

SHAN, L.; ZHU, H. Camle: A caste-centric agent-oriented modelling language and environment. In: . [*S. l.*: *s. n.*], 2004. v. 3390, p. 144–161. ISBN 978-3-540-24843-9.

SOMMERVILLE, I. **Engenharia de software**. 9ª. ed. [*S. l.*]: Pearson, 2011.

Tilkov, S.; Vinoski, S. Node.js: Using javascript to build high-performance network programs. **IEEE Internet Computing**, v. 14, n. 6, p. 80–83, 2010.

VILLAMIZAR, M.; GARCéS, O.; OCHOA, L.; CASTRO, H.; SALAMANCA, L.; MERINO, M. V.; CASALLAS, R.; GIL, S.; VALENCIA, C.; ZAMBRANO, A.; LANG, M. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: . [*S. l.*: *s. n.*], 2016.

Xu, C.; Zhu, H.; Bayley, I.; Lightfoot, D.; Green, M.; Marshall, P. Caople: A programming language for microservices saas. In: **2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)**. [*S. l.*: *s. n.*], 2016. p. 34–43.