

Aritana Noara Costa Santos

Rastreamento Automático Da Causa Raiz Da Pilha De Exceções Em Microserviços Java

Belo Horizonte

2022

Aritana Noara Costa Santos

Rastreamento Automático Da Causa Raiz Da Pilha De Exceções Em Microsserviços Java

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação.

Centro Federal de Educação Tecnológica de Minas Gerais – CEFET-MG

Departamento de Computação

Curso de Engenharia da Computação

Orientador: Prof. Dr. Eduardo Cunha Campos

Belo Horizonte

2022

Centro Federal de Educação Tecnológica de Minas Gerais

Curso de Engenharia de Computação
Avaliação do Trabalho de Conclusão de Curso

Aluno: Aritana Noara Costa Santos

Rastreamento Automático Da Causa Raiz Da Pilha De Exceções Em Microserviços Java

Data da defesa: 01/07/2022

Horário: 10:00

Local da defesa: CEFET-MG Campus II à Avenida Amazonas 7675. Prédio 17 (DECOM), sala 401.

O presente Trabalho de Conclusão de Curso foi avaliado pela seguinte banca:

Professor Doutor Eduardo Cunha Campos — Orientador
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais

Professora Doutora Kecia Aline Marques Ferreira — Membro da banca de avaliação
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais

Professor Doutor Edson Marchetti Da Silva — Membro da banca de avaliação
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais

*Dedico esse trabalho a Deus, aos meus pais: José Firmino da Costa e Marina
Alves dos Santos, família, professores e meus grandes amigos.*

Agradecimentos

Agradeço a todos que me apoiaram, que me fizeram notar meus muitos erros e perceberam alguns dos meus acertos. Percebo, no decorrer dos anos que errar... errar é algo tão comum que seria muito valioso se talvez possamos utilizar de tratamentos de exceções, testes e melhoria contínua para nossa própria existência, mas não sou o primeiro a pensar sobre o assunto. Lembro-me de Antoine Saint-Exupery, quando certa vez escrevera: quando, por mutação, nasce uma rosa nova nos jardins, os jardineiros se alvoroçam. A rosa é isolada, é cultivada, é favorecida, mas não há jardineiros para os homens.

“É do senso comum capturar um método e experimentá-lo. Se ele falhar, admita isso com franqueza e experimente outro. Mas acima de tudo, tente algo.”
(Franklin Delano Roosevelt)

Resumo

Uma aplicação de microsserviços consiste em pequenos serviços independentes que se comunicam usando APIs bem definidas. É comum que aplicações utilizem mensagens de log para monitorar o sistema. Um serviço raramente realiza o log de uma exceção, que pode ser a causa raiz de uma falha, por isso é importante a identificação rápida, e o registro de exceções em arquivo de logs não se mostra ideal pois exceções são compostas de múltiplas linhas. A hipótese a ser verificada nesse estudo é que quando uma requisição não é bem sucedida e gera uma exceção em um determinado serviço, seria possível rastrear o escopo onde a exceção inicial foi lançada, a causa raiz. Se for utilizada a técnica de exceções encadeadas, identificar cada instância da requisição com um identificador único e salvar a pilha de exceções um serviço central, então é possível investigar se uma aplicação pode gerar automaticamente a resolução da falha. A revisão literária revela que exceções são geralmente registradas em arquivos de log, com um identificador único, em um serviço centralizado para logs. A análise da exceção depende de ferramentas para coleta, recuperação e visualização de logs, como a pilha ELK: Logstash, Elasticsearch e Kibana. Há abordagens que analisam o código estaticamente para construir gráficos de propagação de exceções para auxiliar os desenvolvedores. Outras propostas coletam diversas métricas para geração de grafos de impacto. O mercado oferece algumas ferramentas de monitoramento de exceções, como a aplicação Sentry, contudo, com ênfase à coleção de exceções, mas não efetua a classificação e a resolução da falha. Ao contrário das soluções atuais, que exigem do usuário experiência adequada do projeto, a proposta desse trabalho é apresentar um estudo de caso para monitoramento de exceções de software de maneira automatizada, com *interface* simples, dependente de tratamento adequado de exceção, para bom funcionamento. Esse trabalho é limitado à fase de projeto e desenvolvimento em ambiente local.

Palavras-chave: Exceções encadeadas, microsserviços, Spring Boot, Java, rastreamento automático, causa raiz, pilha de exceção.

Abstract

A microservices application consists of small independent services that communicate using well-defined APIs. It is common for applications to use log messages to monitor the system. A service rarely logs an exception, which could be the root cause of a failure, so it is important to quickly identify, and the recording of exceptions in the log file is not ideal because exceptions are composed of multiple lines. The hypothesis to be verified in this study is that when a request is not successful and throws an exception on a given service, it would be possible to trace the scope where the initial exception was thrown, the root cause. If using the technique of chained exceptions, identify each instance of the request with a unique identifier and save the exception stack to a central service, then it is possible to investigate whether an application can automatically generate the resolution of the failure. Literary review reveals that exceptions are usually recorded in archives log, with a unique identifier, in a centralized service for logs. The analysis of the exception depends on tools for collecting, retrieving and viewing logs, like the ELK stack: Logstash, ElasticSearch and Kibana. There are approaches that analyze the code statically to build exception propagation graphs to assist the developers. Other proposals collect different metrics to generate impact graphs. The market offers some monitoring tools for exceptions, such as the Sentry application, however, with an emphasis on the collection of exceptions, but does not perform fault classification and resolution. Unlike current solutions, that demand adequate project experience from the user, the proposal of this work is to present a case study for monitoring software exceptions from automated way, with simple interface, dependent on proper treatment exception, for proper functioning. This work is limited to the design phase and development in local environment.

Keywords: Chained Exceptions, microservices, spring Boot, Java, automatic tracing, initial cause, exception stack.

Lista de figuras

Figura 1 – A ilustração de URL, URN e URI	30
Figura 2 – Monolito modular	32
Figura 3 – Três serviços realizando chamadas síncronas	34
Figura 4 – <i>API Gateway</i> com <i>Service Discovery</i>	34
Figura 5 – Subconjunto do conjunto hierárquico de herança da classe <i>Throwable</i>	39
Figura 6 – Lançamento de uma exceção capturada em diferentes escopos <i>Throwable</i>	40
Figura 7 – Uma cadeia de exceções encadeadas	40
Figura 8 – Serviços Loja, Fornecedor Estadual e Fornecedor Municipal	47
Figura 9 – Cenário de teste por simulação de requisição de cliente via Postman	48
Figura 10 – Injeção de traceId na requisição.	50
Figura 11 – Bloco try...catch: exceção encadeada.	51
Figura 12 – Pilha de exceção é tratada e enviada ao MongoDB Atlas.	52
Figura 13 – Aplicação para resolver e gerar informações de falhas no sistema.	53
Figura 14 – Modelagem de pacotes Java	58
Figura 15 – Pacote config	58
Figura 16 – Pacote Controller	59
Figura 17 – Pacote Service	60
Figura 18 – Pacote Dto	60
Figura 19 – Pacote Exception	61
Figura 20 – Pacote Model	61
Figura 21 – Pacote NetWorking	62
Figura 22 – Application.properties	63
Figura 23 – Application.properties: Configura a aplicação.	63
Figura 24 – TraceID: ID único da requisição/resposta HTTP	65
Figura 25 – Bloco Exception Handler: Onde todas as exceções são observadas.	66
Figura 26 – Resultado para uma requisição de teste	68
Figura 27 – TrackerCentral	69

Figura 28 – <i>Schema</i> do objeto Exceção	70
Figura 29 – TrackerCentral	71
Figura 30 – TrackerCentral	72
Figura 31 – ExceptionTracker: lista de execuções.	74
Figura 32 – ExceptionTracker: seleção de exceção para verificar sua causa raiz.	74
Figura 33 – ExceptionTracker: exceções encadeadas com causa raiz em des- taque.	74
Figura 34 – ExceptionTracker: link para acessar o escopo no GitHub.	75
Figura 35 – ExceptionTracker: avaliação de falha no serviço Fornecedor Es- tadual.	78
Figura 36 – ExceptionTracker: requisição com erro de grafia no campo de endereço do estado.	79
Figura 37 – ExceptionTracker: erro de resposta da requisição.	79
Figura 38 – ExceptionTracker: busca pelo traceId df6680cdc5a20612.	80
Figura 39 – ExceptionTracker: escopo da falha.	80
Figura 40 – ExceptionTracker: link do código no GitHub.	81
Figura 41 – ExceptionTracker: código no GitHub.	81
Figura 42 – Bloco try...catch: exceção encadeada.	83
Figura 43 – CentralExceptionHandler: captura e salva a exceção no MongoDB.	84

Lista de tabelas

Tabela 1 – Informações que identificam um serviço. Fonte: Elaborada pelo autor.	47
Tabela 2 – Grupo de exceções utilizadas para serem causa raiz na fase de testes. Fonte: Elaborada pelo autor.	48
Tabela 3 – Componentes da arquitetura e responsabilidades.	55
Tabela 4 – Serviços e responsabilidades.	56
Tabela 5 – Configuração ambiente de desenvolvimento.	56

Lista de abreviaturas e siglas

API	Application Programming Interface
ELK	Logstash, Elasticsearch e Kibana
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
REST	Representational State Transfer
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
WEB	Word Wide Web

Sumário

1	INTRODUÇÃO	25
2	REFERENCIAL TEÓRICO	29
2.1	Arquitetura de Software	29
2.1.1	Arquitetura <i>Web</i>	29
2.1.1.1	Recursos e Identificadores na <i>Web</i>	30
2.1.2	Estilo de Arquitetura REST	31
2.1.3	Arquitetura em Nuvem	31
2.1.4	Arquitetura Monolítica	32
2.1.5	Arquitetura de Microserviços	32
2.2	Comunicação	33
2.3	Computação em Nuvem	34
2.4	Rastreamento Distribuído	35
2.5	HTTP	36
2.6	API	36
2.6.1	API Rest	36
2.7	Spring Boot	37
2.8	Postman	37
2.9	MariaDB	37
2.10	MongoDB	38
2.11	Tratamento de Exceções em Java	38
2.11.1	Exceções Encadeadas	38
3	TRABALHOS RELACIONADOS	41
4	METODOLOGIA	45
4.1	Características	45
4.2	Planejamento	46
4.2.1	Microserviços	46
4.2.1.1	Serviços e Regra de Negócio	46

4.2.1.2	Número de Porta	47
4.2.1.3	Classes de Exceções Adotadas para Testes	47
4.2.2	ExceptionTracker	49
4.2.2.1	Back-end ExceptionTracker	49
4.2.2.2	Front-end ExceptionTracker	49
4.3	Implementação	49
4.3.1	Microserviços	49
4.3.1.1	Injetar Identificador Único da Transação — TracelD	49
4.3.1.2	Registrar Pilha de Exceções no MongoDB	50
4.3.2	ExceptionTracker	51
5	DESENVOLVIMENTO	55
5.1	Microserviços	56
5.1.1	Ambiente de Desenvolvimento	56
5.1.2	Bibliotecas Utilizadas	56
5.1.3	Arquitetura	57
5.1.3.1	Config	58
5.1.3.2	Controller	59
5.1.3.3	Service	59
5.1.3.4	Dto	60
5.1.3.5	Exception	61
5.1.3.6	Model	61
5.1.3.7	NetWorking	62
5.1.3.8	Configuração da Aplicação	62
5.1.3.9	Rotas	64
5.1.4	Principais Funcionalidades Implementadas	64
5.2	Back-end ExceptionTracker	69
5.2.1	Conexao MongoDB	69
5.2.2	<i>Schema</i> de persistência no banco de dados	69
5.2.3	Rotas	70
5.2.4	Principais Funcionalidades Implementadas	70
5.3	Front-end ExceptionTracker	71
5.3.1	Componentes	71

5.3.2	Rotas	72
5.3.3	Views	72
5.3.4	Principais Funcionalidades Implementadas	73
5.3.5	Informações Acerca do Projeto de Estudo de Caso	76
6	AVALIAÇÃO	77
6.1	Falha na Regra de Negócio	77
7	LIMITAÇÕES DA FERRAMENTA	83
8	CONCLUSÕES E TRABALHOS FUTUROS	87
	REFERÊNCIAS	89

1 Introdução

De acordo com [Richardson \(2019\)](#) uma aplicação consiste em múltiplos serviços e instâncias de serviço executados em várias máquinas. Ao lidar com uma requisição, algum erro pode ocorrer e uma instância de serviço pode lançar uma exceção. É comum que as aplicações utilizem mensagens de *log* para registrar e auxiliar o monitoramento do sistema. Um microsserviço raramente realiza o *log* de uma exceção, mas quando uma exceção ocorre é importante a identificação da causa raiz, pois uma exceção pode ser um sintoma de uma falha do programa.

[Chen, Li e Li \(2017\)](#) descrevem que a Arquitetura Monolítica é mais utilizada para desenvolvimento de software cuja principal característica é encapsular todas as funções em uma única aplicação. [Fowler\(2022\)¹](#) realça que o termo monolítico foi muito utilizado na comunidade *Unix* para denotar sistemas que ficam enormes. Todavia, em 2011, próxima à cidade de Veneza, no *Workshop Of Software Architects*, foi definido um nome para uma nova forma de criar aplicações:Arquitetura de Microsserviços.

Um dos fatores para o aumento da popularidade dessa nova arquitetura, conforme [Lauretis \(2019\)](#) seria justamente o avanço da computação em nuvem, pois das arquiteturas que se beneficiam dessa infraestrutura, a Arquitetura de Microsserviços é a que se mostra mais relevante. [Fowler\(2022\)¹](#) explica que com essa arquitetura é possível desenvolver uma única aplicação como um conjunto de pequenos serviços, cada qual trabalha em seu próprio processo e consegue ser implantado automaticamente, ao contrário de aplicações monolíticas nas quais uma alteração, mesmo que em uma pequena parte do sistema, implica em uma reconstrução e outra implantação. [Carnell e Sanchez \(2021\)](#) citam que cada serviço individual pode ser empacotado em uma máquina virtual em um provedor na nuvem e a partir dessa, diversas outras instâncias podem ser criadas, utilizando o conceito de escalabilidade horizontal, outra característica da Arquitetura de Microsserviços.

¹ Disponível em: <<https://martinfowler.com/articles/microservices.html>>. Acesso em: 5 fev. 2022.

Carnell e Sanchez (2021) definem algumas principais desvantagens, em relação à Arquitetura Monolítica, a dificuldade de depuração e de rastreamento do estado dos serviços.

Em uma pesquisa recente, Zhou et al. (2018) destacam que análise de *log*, em formato de relatórios, para análise de falha e depuração é amplamente utilizada por desenvolvedores na indústria de software de microsserviços. Segundo Zhou et al. (2018), os desenvolvedores necessitam interpretar as informações recebidas, e conforme sua experiência, concebem julgamentos preliminares da causa raiz para tomar decisão de depurar e reproduzir a falha. Após configurar o ambiente e reproduzir a falha, os profissionais buscam identificar o escopo, em outras palavras, em qual serviço e em, qual parte do código a causa raiz possa estar localizada. Após encontrar a falha raiz, é efetuada a correção e testes.

Na arquitetura de microsserviços, já existem padrões de resiliência, que prescrevem criar artefatos para evitar que um serviço com defeito possa prejudicar o desempenho do sistema, descrito por Carnell e Sanchez (2021). Essa questão é bem alinhada à técnica *fail fast*, que sugere que sistemas que falham imediata e visivelmente tendem a ser mais robustos, porque na cadeia de testes, os erros podem ser encontrados facilmente e limitando-os de chegarem à fase de produção, como declarado por Shore (2004).

Para Richardson (2019) um serviço raramente realiza o *log* de uma exceção. Quando a exceção ocorre é importante que se identifique a causa raiz, por ser uma erro na execução do programa. Esse mesmo trabalho afirma que o modo tradicional de inspecionar arquivos de *logs* para verificar se houve exceções, não é o ideal, por não ser ágil e apresentar inconvenientes como não ser a melhor estrutura de armazenar dados da pilha de uma exceção, e na possibilidade de haver exceções duplicadas não há mecanismo para tratá-las como únicas.

Dessa forma, nessa monografia apresentamos uma abordagem para rastrear exceções em microsserviços, com foco na exceção inicial e automaticamente. Isso é feito por meio da injeção de um identificador único para uma requisição que percorre múltiplos microsserviços, e da coleta do escopo da falha, e com essas informações salvas em um banco *NoSQL*, para posteriormente uma aplicação

executar consultas nesse banco de dados. Diante disso, faz sentido presumir que gerar automaticamente um relatório com o escopo da falha para ser utilizado pelo desenvolvedor, independente de seu grau de maturidade técnica, como apoio para solucionar a falha com maior agilidade é algo factível de ser realizado.

A revisão literária, ainda revela, que existem ferramentas poderosas para auxiliar os desenvolvedores a extrair informações de arquivos de log, a título de exemplo, as ferramentas: Logstash, Elasticsearch e Kibana, que constituem a denominada pilha ELK. Com essa pilha é possível criar filtros e gerar visualizações, como cita [Newman \(2015\)](#). Há abordagens que analisam o código estaticamente para construir gráficos de propagação de exceções para auxiliar os desenvolvedores, como descrito por [Fu e Ryder \(2007\)](#), outras propostas utilizam métricas coletadas, como latência, para gerar grafos de impacto, destacado por [Pina et al. \(2018\)](#). Considerando ferramentas de monitoramento de exceções, [Lenarduzzi et al. \(2017\)](#) ao abordar o tema, cita algumas, como as recentes aplicações Sentry, HoneyBadger e Exceptionless. Essas ferramentas têm o objetivo de monitorar continuamente as exceções de software, mas ainda há espaço de pesquisa para melhorar a predição e classificação de exceções para auxiliar os desenvolvedores.

[Zhou et al. \(2018\)](#) descrevem que a maioria das companhias prefere implementar ferramentas de rastreamento e visualização próprias por questões de implementação das técnicas de arquitetura de microserviços e vai além ao dizer que para utilizar as ferramentas, o nível de maturidade técnica e experiência no projeto são relevantes para executar bem, uma tarefa de análise de falhas.

O principal objetivo da ferramenta é permitir que o desenvolvedor encontre o escopo e a causa raiz da exceção, e por não ser necessário efetuar análises de arquivos de logs, na fase de análise de escopo de falha, um benefício gerado é ganho de tempo na manutenção do sistema que a ferramenta analisa. As exceções salvas em banco de dados são pesquisadas por um serviço em Node.js que acessa o banco de dados e serve como *back-end* de uma aplicação implementada em Vue.js que trabalha como *interface* gráfica para este estudo. Nesse trabalho, a abordagem se limita a etapa de projeto e desenvolvimento dos serviços, e seria uma sugestão de tema futuro, a análise na perspectiva da fase de produção, com máquinas virtuais distribuídas na arquitetura de nuvem, em um projeto real.

O restante deste trabalho está organizado da seguinte maneira. O Capítulo 2 apresentar os principais conceitos relacionados ao trabalho. O Capítulo 3 aborda pesquisas e temas que outros autores realizaram. O Capítulo 4 visa esclarecer de que forma foi planejado este estudo em relação à sua construção para então ser demonstrado no Capítulo seguinte. O Capítulo 5 é possível ter uma visão acerca de todo o projeto, arquitetura, motivações e implementações. O Capítulo 6 permite avançar nos aspectos de validação da construção tecnológica deste estudo, explica como se obteve o resultado. Logo após, o Capítulo 7 exhibe os pontos de limitação da ferramenta ExceptionTracker, o produto desse trabalho.

2 Referencial Teórico

Nesse Capítulo será apresentado o referencial teórico para fornecer antecedentes sobre o tema em estudo, além de informações sobre os aspectos metodológicos para a concepção da parte prática do trabalho de conclusão de curso.

2.1 Arquitetura de Software

Não há uma definição muito clara para o termo “Arquitetura de Software” na literatura. O Instituto de Engenharia de Software da Carnegie Mellon University compilou uma lista¹ de referências bibliográficas que definem o termo arquitetura de software, de onde é possível observar que diversos autores utilizam o termo “arquitetura de software” para retratar a maneira como é organizado um sistema. Esse conceito se relaciona a esse trabalho na definição do termo “Arquitetura de Microserviços”.

2.1.1 Arquitetura *Web*

Webber, Parastatidis e Robinson (2010) relatam que Tim Berners-Lee desenvolveu as fundações da *Web* (*World Wide Web*) na década de 90 com a intenção de viabilizar um sistema para compartilhar documentos que tivesse algumas características como: ser distribuído, ser fracamente acoplado e ser fácil de ser utilizado. Pollard (2019) faz a distinção entre a *Web* e a internet. Nas palavras de Pollard, a internet é uma coleção de computadores ligados em rede e a *Web* é um dos serviços que operam na internet. MDN WEB Docs² explica que dispositivos conectados à *web* são chamados clientes e servidores. Clientes são dispositivos que solicitam recursos a outro dispositivo, o servidor. Esses recursos podem ser documentos, sites, etc. Para que os dispositivos se comuniquem, existem regras

¹ Disponível em: <<https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513807>>. Acesso em: 5 fev. 2022.

² Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/Getting_started_with_the_web/How_the_Web_works>. Acesso em: 9 fev. 2022.

que recebem o nome de protocolos. O protocolo mais comumente utilizado na *Web* é o protocolo de Transferência de *Hypertexto* (*Hypertext Transfer Protocol*, ou simplesmente HTTP). Esse conceito é importante para esse trabalho por ser os microserviços implantados em servidores fornecedores de recursos aos clientes na rede de internet e cuja comunicação utiliza o protocolo HTTP.

2.1.1.1 Recursos e Identificadores na Web

Para Webber, Parastatidis e Robinson (2010) os recursos são os blocos fundamentais de construção de sistemas *Web*. Recursos podem ser definidos como qualquer coisa exposta na *Web*, como um vídeo ou uma imagem. Recursos necessitam ser identificados e manipulados e para esses propósitos existe o *Uniform Resource Identifier* (URI), que seria um identificador de recursos uniforme que permite endereçar unívoca e completamente um recurso e também ser manipulado por protocolos de aplicação como o HTTP. O *Uniform Resource Locator* (URL) é um localizador uniforme de recursos e o *Uniform Resource Name* (URN) é o nome uniforme de um recurso. Ambos são parte do subconjunto do URI, e seu objetivo é identificar onde o recurso está disponibilizado como visa ilustrar a Figura 1.³

Figura 1 – A ilustração de URL, URN e URI



Fonte: uriTo (2022)

Essa informação é útil para esse estudo porque os microserviços expõem as funcionalidades de sua API através de URLs que identificam os recursos.

³ Disponível em: <<https://uri.to/faq.php>>. Acesso em: 9 fev. 2022.

2.1.2 Estilo de Arquitetura REST

Como abordado por ALLET (2013)⁴ *Representational State Transfer* (REST), é um estilo arquitetural que define princípios para que recursos *Web* sejam definidos e endereçados. Esse tudo tem como principais características:

- **Cliente-Servidor:** Separação de responsabilidades entre cliente e servidor de forma permitir a evolução independente.
- **Interface Uniforme:** Os recursos devem ser identificados e representados. As mensagens devem ser auto-descritivas e a *Application Programming Interface* (API), deve fornecer *links* que indicarão aos clientes como navegar através desses recursos.
- **Sem estado:** O estado de comunicação não é mantido no servidor.
- **Cache:** Requisições do cliente podem ser armazenadas em *caches* para otimizar solicitações futuras.
- **Camadas:** A arquitetura deve possuir camadas independentes.

Os microsserviços implementam APIs RESTful, termo designado para implementações que seguem o estilo arquitetural REST.

2.1.3 Arquitetura em Nuvem

GEEKSFORGEES (2022)⁵ define computação em nuvem como o termo designado para descrever o acesso e armazenamento de dados e programas que ocorre em servidores remotos. A arquitetura em nuvem diz respeito aos componentes requeridos para a computação em nuvem e seus arranjos.

A arquitetura em nuvem não foi utilizada nesse trabalho, mas sua menção se fez por ser majoritariamente utilizada na implantação dos servidores de microsserviços em ambiente de produção.

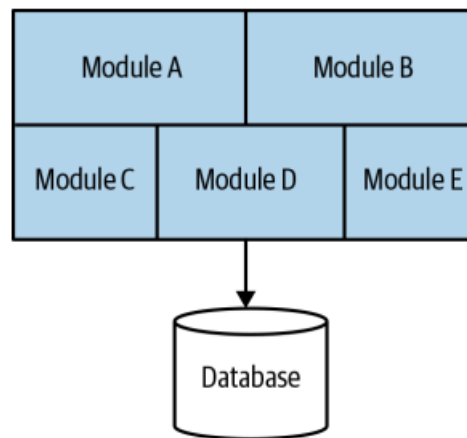
⁴ Disponível em: <<https://www.devmedia.com.br/introducao-ao-rest-ws/26964>>. Acesso em: 9 fev. 2022.

⁵ Disponível em: <<https://www.geeksforgeeks.org/cloud-computing>>. Acesso em: 9 fev. 2022.

2.1.4 Arquitetura Monolítica

De acordo com Newman (2015) sistemas que possuem uma arquitetura monolítica, no fluxo de desenvolvimento, devem ser implantados de forma única, em outras palavras, pequenas porções do sistema não podem ser implantadas independentemente. Essa arquitetura possui algumas vantagens em relação à arquitetura de Microserviços, como fluxo de desenvolvimento, testes *end to end* e monitoramento simples, bem como a facilidade de reuso de software. Na Figura 2 há uma representação de um sistema monolítico modular onde cada módulo é independente, porém todos os módulos devem ser combinados na fase de implantação.

Figura 2 – Monolito modular



Fonte: newman, wilding, odate.

Essa seção contribui para esse trabalho como informação acerca da evolução da arquitetura utilizada.

2.1.5 Arquitetura de Microserviços

De acordo com Newman (2015) aplicações de arquitetura de microserviços surgiram como uma tendência ou padrão para seguir na evolução de novas concepções de construção de software que a indústria tem experimentado como: *Domain-driven design* (foco no domínio da aplicação) entrega contínua, virtualização por demanda, automação de infraestrutura, pequenas equipes de trabalho

autônomos e escalabilidade de sistemas. Algumas das características dos serviços das aplicações de arquitetura de microsserviços, destacam-se:

- **Pequenos e focados em fazer bem uma responsabilidade:** as fronteiras dos serviços devem coincidir com as fronteiras do negócio.
- **Autônomos:** o serviço deve ser uma entidade separada, capaz de ser implementada como um serviço isolado.
- **Heterogeneidade de tecnologias:** um serviço pode utilizar tecnologia única e independe da tecnologia aplicada em outro serviço, ou seja, a implementação é encapsulada.
- **Resilientes:** existem técnicas que evitam falhas encadeadas, permitindo que o sistema isole o problema do restante da aplicação.
- **Escalabilidade:** cada serviço pode ser escalado independentemente quando houver necessidade.
- **Facilidade de implantação:** cada serviço pode ser criado, refatorado e então implantado independentemente dos demais.

Esses conceitos são úteis ao trabalho para a compreensão das principais características dos microsserviços.

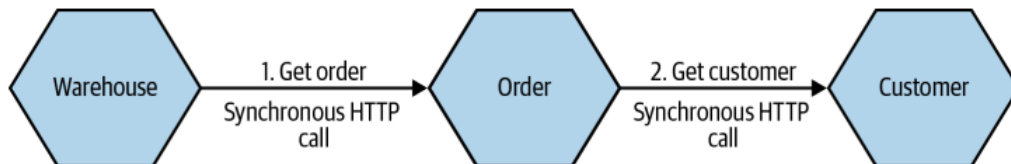
2.2 Comunicação

De acordo com [Richardson \(2019\)](#) os microsserviços são fracamente acoplados. Toda interação com um serviço acontece por sua API, que encapsula os detalhes de sua implementação. A Figura 3 ilustra uma aplicação de serviços se comunicando sincronicamente.

[Carnell e Sanchez \(2021\)](#) declaram que uma instância de serviço deve fazer seu registro em um serviço de descoberta (*service discovery*) que no que lhe concerne, registra o endereço IP (*Internet Protocol*) ou endereço de domínio e nome lógico. O cliente então se comunica com o serviço de descoberta que se encarrega

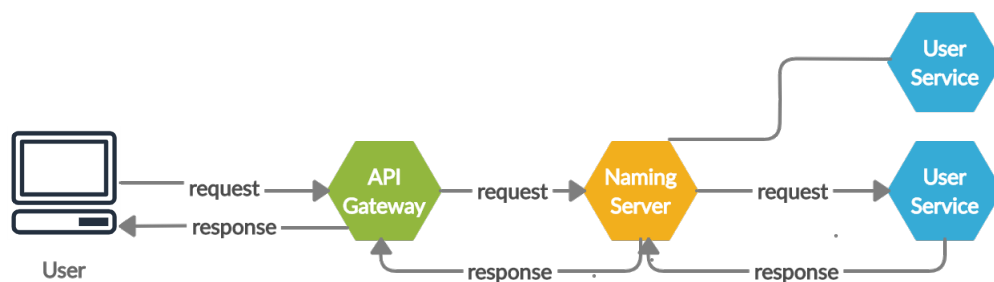
de redirecionar a requisição para o serviço correto. [Richardson \(2019\)](#) também menciona que um serviço chamado *API Gateway* é necessário para lidar com falhas de serviços, no contexto de resiliência, e distribuição de carga de trabalho. A Figura 4 ilustra os serviços *API Gateway* e *Service Discovery*.

Figura 3 – Três serviços realizando chamadas síncronas



Fonte: [Newman \(2015\)](#).

Figura 4 – *API Gateway* com *Service Discovery*



Fonte: [PRIYA \(2020\)](#).

Os serviços implementados nesse trabalho se comunicam através de suas APIs. Nesse trabalho não é utilizado o conceito de *API Gateway* e *Service Discovery*. Em um trabalho futuro, com implantação dos serviços na arquitetura de nuvem, esse conceito pode ser importante.

2.3 Computação em Nuvem

[Carnell e Sanchez \(2021\)](#) afirma que microserviços podem ser implantados em imagens de máquinas virtuais, que virtualizam camadas de hardware, ou

em contêineres virtuais, que virtualizam camadas de software,⁶ e são executados dentro de uma máquina virtual em servidores remotos. Uma grande vantagem da computação em nuvem é o conceito de elasticidade, ou seja, provedores de serviço *Cloud* permitem que rapidamente novas máquinas virtuais, e contêineres, sejam implantados conforme demanda de carga de serviço. Por isso que são necessários serviços como *API Gateway* para fazer o balanço da carga para requisições e *Service Discovery* para fazer o balanço da rota interna de cada serviço da aplicação, descrito na Subseção anterior 2.2.

Essa seção é importante para o trabalho, pois aborda como a arquitetura de microserviços é atualmente implantada.

2.4 Rastreamento Distribuído

Carnell e Sanchez (2021) informa que microserviços são distribuídos, e a depuração para encontrar uma falha é algo muito difícil de ser realizado, pois, implica em rastrear uma ou mais transações através de vários serviços e agregar as informações para inferir algum sentido útil para encontrar a falha. Já existem alguns padrões na indústria para resolver essa questão:

- **Identificadores relacionados:** é possível identificar uma única transação que percorre vários serviços com auxílio da biblioteca Spring Cloud Sleuth. Esse identificador é comumente referido como *traceId*.
- **Agregação de logs :** reunir todos os *logs* da aplicação em um único banco de dados para permitir pesquisas simples.
- **Visualização de fluxo:** visualizar o fluxo das transações entre serviços bem como disponibilizar dados de desempenho em cada parte da sequência da transação. Para tal existe uma biblioteca Zipkin que permite a visualização de dados e do fluxo da transação entre serviços.

⁶ Disponível em: <<https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms>>. Acesso em: 7 jul. 2022.

Os padrões: identificadores relacionados e Agregação de *logs* serão utilizados nesse trabalho. Os serviços utilizam a biblioteca Spring Cloud Sleuth, responsável por injetar na transação entre serviços um identificador único. Os dados de *logs* serão salvos em um banco de dados centralizado, MongoDB Atlas, sendo, portanto, o banco que agrega dados de exceções lançadas por todos os serviços.

2.5 HTTP

O Protocolo de Transferência de Hipertexto (HTTP) é o protocolo da camada de aplicação da *Web* que estabelece as regras de comunicação entre dois programas, cliente e servidor. O HTTP utiliza o protocolo de controle de transmissão (TCP), como protocolo de transporte como Kurose e Ross (2013) definem.

2.6 API

A Application Programming Interface (API), que pode traduzido como Interface de Programação de Aplicações, se caracteriza por ser uma coleção de protocolos de comunicação e sub-rotinas que permitem programas comunicarem entre si (GEEKSFORGEEKS, 2022)⁷. Lauret (2019) define API como *interface* exposta por algum software, abstraindo a complexidade da implementação. Cada serviço possui sua API por onde ocorre a interação com outros serviços.

2.6.1 API Rest

Uma aplicação Rest API, de acordo com Lauret (2019), utiliza do estilo arquitetural REST para atingir seu objetivo de ser uma *interface* de comunicação. Portanto, utiliza protocolo para comunicação, identifica recursos e relações, identifica ações, parâmetros e retorno, bem como designa caminho de recursos.

⁷ Disponível em: <<https://www.geeksforgeeks.org/introduction-to-apis>> Acesso em: 5 fev. 2022.

2.7 Spring Boot

De acordo com Carnell e Sanchez (2021) Spring⁸ é um *framework* de construção de aplicações Java, que possui o conceito de injeção de dependências que permite externalizar relacionamentos entre objetos por convenções e anotações de código, evitando assim que os objetos necessitem ter conhecimento de outros implicitamente, por código. Spring Boot⁹ é uma transformação que agrega o núcleo do framework Spring de forma que entrega uma solução baseada em Java, orientada a REST. Com simples anotações um desenvolvedor pode rapidamente construir um serviço REST que pode ser implantado sem necessidade de um contêiner. Para Carnell e Sanchez (2021) Spring Boot viabiliza criação de aplicações REST, seja simplificando o mapeamento de estilos de verbos (HTTP) para URLs e serialização de protocolo JavaScript Object Notation (JSON)¹⁰ e principalmente, para o foco desse trabalho, mapear exceções (Java) para códigos de erros no padrão HTTP.

2.8 Postman

Postman é uma plataforma de construção e utilização de APIs.¹¹ A ferramenta permite testar serviços RESTful por meio do envio de requisições HTTP e da análise do seu retorno.¹² Nesse trabalho, o Postman serviu para testar as APIs do *back-end* da interface gráfica bem como as APIs dos microsserviços.

2.9 MariaDB

MariaDB é um banco de dados relacional *OpenSource*, substituto do banco de dados MySQL. Utiliza a linguagem padrão para realizar consultas, *Standard Query Language* (SQL).¹³ Nesse trabalho utilizaremos MariaDB, para o registro de dados provenientes do modelo de negócio dos microsserviços.

⁸ Disponível em: <<https://spring.io/>> Acesso em: 6 fev. 2022.

⁹ Disponível em: <<https://start.spring.io/>> Acesso em: 6 fev. 2022.

¹⁰ Disponível em: <<https://www.json.org/json-en.html>> Acesso em: 6 jul. 2022.

¹¹ Disponível em: <<https://www.postman.com>> Acesso em: 6 fev. 2022.

¹² Disponível em: <<https://www.devmedia.com.br/testando-apis-web-com-o-postman/37264>> Acesso em: 6 jul. 2022.

¹³ Disponível em: <<https://mariadb.org>> Acesso em: 6 fev. 2022.

2.10 MongoDB

MongoDB¹⁴ é um banco de dados não relacional, de documentos com escalabilidade e flexibilidade. Ele permite armazenar dados em documentos do tipo JSON, permitindo flexibilidade para alterar os campos de documento para documento, bem como a estrutura. A opção MongoDB utilizada nesse trabalho permite conectar a aplicação à sua API na nuvem, gratuitamente, mas com limitação de 512 MB de armazenamento.

2.11 Tratamento de Exceções em Java

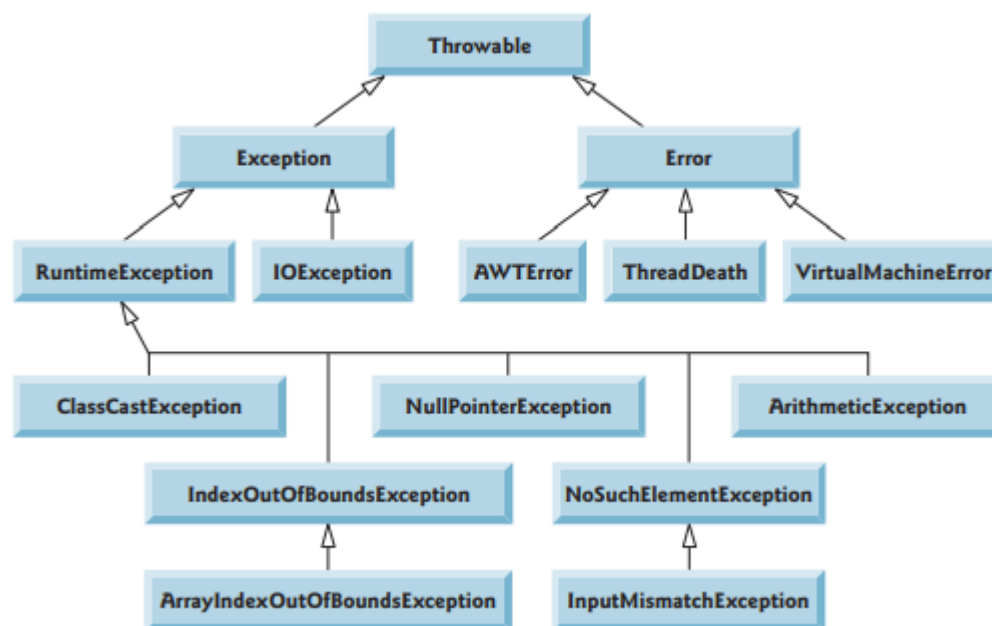
DEITEL e DEITEL (2010) reportam que uma exceção é uma indicação de um problema que ocorre na execução do programa. O tratamento de exceções visa permitir que os programas processem erros de forma síncrona. Em Java, a classe *Exception* e suas subclasses representam as situações excepcionais que podem ocorrer em um programa. A Figura 5 exibe parte da hierarquia de herança da classe *Throwable*.

O Java distingue as exceções não verificadas, ou seja, exceções causadas por defeitos no código e exceções verificadas, causadas por condições que não estão sobre o controle do programa. Exceções verificadas são checadas pelo compilador e é mandatório que o desenvolvedor as capture ou as declare em uma cláusula *throws*, destacado por DEITEL e DEITEL (2010), Nguyen e Sveen (2003) comentam que a cláusula *throws* é utilizada quando a exceção não pode ser capturada no método em que foi gerada, como mostrado na Figura 6. Portanto, ao ser lançada uma exceção de um escopo para outro pode gerar perda de informações da pilha de exceções do escopo corrente, definido por DEITEL e DEITEL (2010).

2.11.1 Exceções Encadeadas

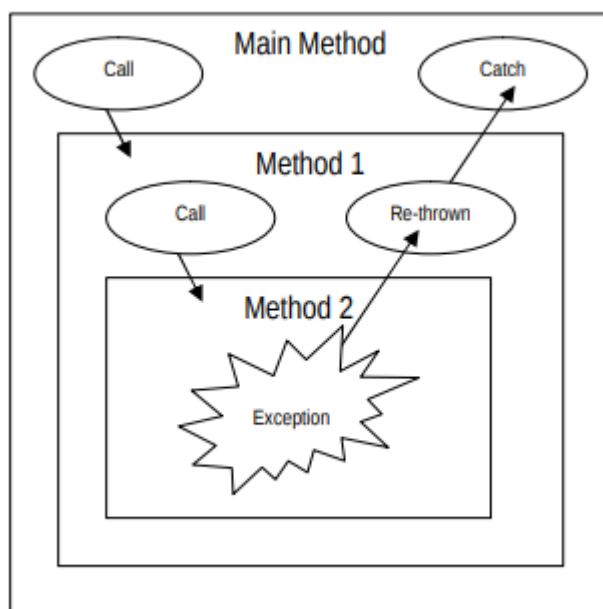
Para evitar a perda de informações da pilha de exceções, ao se utilizar a cláusula *throws* para lançar uma exceção de um escopo para um outro, o Java disponibiliza as exceções encadeadas a partir do Java Development Kit (JDK) 1.4.0,

¹⁴ Disponível em: <<https://www.mongodb.com/cloud/>> Acesso em: 6 fev. 2022.

Figura 5 – Subconjunto do conjunto hierárquico de herança da classe *Throwable*

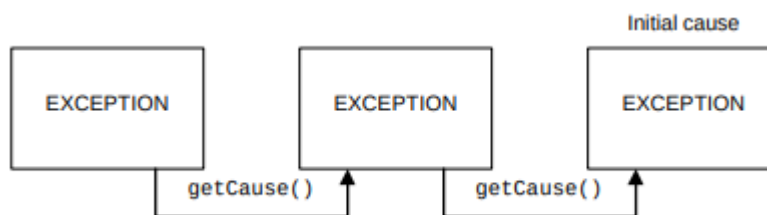
Fonte: [DEITEL e DEITEL \(2010\)](#)

desenvolvido para auxiliar na depuração de falhas, declarado por [Nguyen e Sveen \(2003\)](#) e ilustrado na Figura 7. Vale ressaltar que nesse trabalho utilizaremos o método `getStackTrace()`, próprio das classes de exceção, que disponibiliza o nome da classe, método e linha onde a exceção foi capturada. Do encadeamento das exceções, é obtido dados de todo o escopo de propagação da falha.

Figura 6 – Lançamento de uma exceção capturada em diferentes escopos *Throwable*

Fonte: [Nguyen e Sveen \(2003\)](#).

Figura 7 – Uma cadeia de exceções encadeadas



Fonte: [Nguyen e Sveen \(2003\)](#).

3 Trabalhos Relacionados

Tendo em vista uma abordagem não intrusiva para diagnosticar falhas em serviços, [Pina et al. \(2018\)](#) utilizaram a coleta de *logs* do serviço *Gateway* para avaliar informações das requisições como: IP, porta de origem, porta de destino e tempo de resposta para gerar gráficos da topologia do sistema com adição de informações acerca da latência das requisições e frequência.

[Zhou et al. \(2018\)](#) abordaram em suas pesquisas as ferramentas de depuração de microserviços disponíveis no mercado e as atuais formas de depuração e desafios enfrentados pelos desenvolvedores. Para tal, desenvolveram um estudo de caso com serviços e utilizaram a pilha Elasticsearch + Logstash + Kibana (ELK), o software Zipkin e análise de arquivos de *log*. Concluíram que há necessidade de ferramentas de registros e visualização mais inteligentes, pois as existentes dependem muito da experiência do desenvolvedor para inferir se o atual estado de microserviços pode revelar uma falha.

Ao analisar e revisar as diretrizes básicas e padrões de tratamento de exceções de [Wirfs-Brock \(2006\)](#), que indica em seu estudo que o uso adequado do tratamento de exceções de falhas é crucial para a compreensão, evolução e refatoração em software.

[Fu e Ryder \(2007\)](#) exibem um estudo, em aplicações de servidores, de exceções encadeadas com enfoque na análise em tempo de compilação do código, cuja análise é realizada estaticamente no código para encontrar múltiplos fluxos de exceções e não apenas segmentos desses. Segundo o estudo, essa abordagem é útil para apresentar a arquitetura de tratamento de exceções, indicar vulnerabilidades no sistema e ajudar encontrar a falha raiz de um dado problema. O resultado desse estudo produz gráficos de dependência que realçam o fluxo de exceções entre componentes.

[Chang, Jo e Her \(2002\)](#) pesquisaram a análise estática de códigos escritos na linguagem Java para estimar exceções e então construir um gráfico de propagação de exceções que inclui origem da exceção, tratador de exceção e os caminhos

propagados, para auxiliar desenvolvedores. Para tal realizaram a análise do código, o registro de: exceções cheçadas, construções de exceções, e conjunto de restrições de escopo das exceções, para gerar a visualização da propagação de exceções graficamente.

Fu et al. (2014) realiza uma análise acerca das práticas de registro em forma de *log* por desenvolvedores em sistemas da Microsoft e também utiliza de questionários para desenvolvedores contribuírem para a pesquisa. Os resultados apontam uma excelente acurácia de práticas abordadas por desenvolvedores.

Kim, Sumbaly e Shah (2013) verificam a possibilidade de encontrar a causa raiz de anomalias em arquiteturas orientadas a serviço, com a implementação de um algoritmo que ranqueia as possíveis causas raízes de anomalias através do histórico e tempo corrente de métricas como latência e taxa de transferência, medidas por sensores. Seus resultados apontam para uma significativa melhoria no apontamento de causa raiz de falha.

Lenarduzzi et al. (2017) citam as ferramentas de monitoramento contínuo dedicadas ao monitoramento de exceções existentes no mercado como Sentry, OverOps, Airbrake, Rollbar, Raygun, Honeybadger, Stackhunter, Bugsnag e Exceptionless. Informa também que poucas dessas ferramentas classificam o problema imediatamente e que não provêm percepções para auxiliar desenvolvedores encontrar causas eficientemente. Para utilizar tais ferramentas é necessário incluir *logs* específicos no código para fazer o registro das exceções.

Jiang, Zhang e Ren (2020) utilizaram uma pilha EKL (Elasticsearch + Logstash + Kibana) para coletar dados de *logs*, agregar e disponibilizar sua análise, inclusive gráfica. Essa abordagem fornece ao desenvolvedor opções para fazer análises das métricas coletadas para auxiliar em sua pesquisa por falhas no sistema.

Com o foco na análise de falhas de sistemas e a busca pela causa raiz, é possível dizer que os trabalhos relacionados relatados nesse capítulo buscam analisar o cenário da arquitetura e as maneiras de obter informações sobre falhas, e como encontrar a sua causa raiz, seja coletando métricas através de registro de *logs* e realizar análise desses registros através de ferramentas. No campo de tratamento de exceções, algumas pesquisas realizaram a análise estática do código

para obter cadeias de exceções e gerar informações de apoio ao desenvolvedor, outras provêm relatórios de exceções continuamente dos pontos de *logs* inseridos em certos pontos no código. O objetivo desse estudo é fazer uma proposição de capturas de exceções encadeadas em tempo de execução e gerar relatório com o máximo de informação acerca do escopo da falha, evidenciando o escopo e a exceção inicial, sendo disponibilizado o *link* para o trecho de código, relativo ao escopo da causa raiz, na plataforma GitHub, assim o desenvolvedor poderá ver o código imediatamente.

Dessa maneira, esse trabalho pode ser mais uma contribuição para essa área de estudo ao lado dos trabalhos encontrados na literatura.

4 Metodologia

A metodologia empregada no presente trabalho tem o intuito de criar procedimentos para tornar a solução tecnológica viável. O projeto completo encontra-se no GitHub¹.

4.1 Características

As principais características que ferramenta proposta deve possuir para conseguir para que o objetivo do trabalho seja alcançado, são:

- **Identificador Único:** injetar um identificador em cada transação. Para tanto é necessário utilizar a biblioteca Spring Cloud Sleuth².
- **Pilha de exceções:** tratar e salvar a pilha de exceções encadeadas que possam ocorrer em algum serviço em um local único, o banco de dados MongoDB Atlas³.
- **Serviço central:** banco de dados MongoDB, um serviço em nuvem, que será útil para armazenar os registros de pilhas de exceções⁴.
- **Interface gráfica:** A aplicação desenvolvida em VUE.js⁵ e visa acessar o serviço central para realizar a resolução da exceção, ou seja, dispor de forma automática e organizada pelo fluxo inicial das exceções capturadas, as informações principais, como os nomes de: serviços, classes, métodos e número da linha, portanto disponibiliza ao desenvolvedor um *link* para a plataforma GitHub, direcionado para a classe raiz onde a exceção foi lançada.

¹ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master>>. Acesso em: 9 fev. 2022.

² Disponível em: <<https://spring.io/projects/spring-cloud-sleuth>> Acesso em: 6 fev. 2022.

³ Disponível em: <<https://www.geeksforgeeks.org/chained-exceptions-java/>> Acesso em: 6 fev. 2022.

⁴ Disponível em: <<https://www.mongodb.com/>> Acesso em: 6 fev. 2022.

⁵ Disponível em: <<https://vuejs.org/>> Acesso em: 6 fev. 2022.

4.2 Planejamento

A etapa de planejamento visa definir o projeto a ser implementado. Maiores detalhes serão exibidos Capítulo 5.

4.2.1 Microserviços

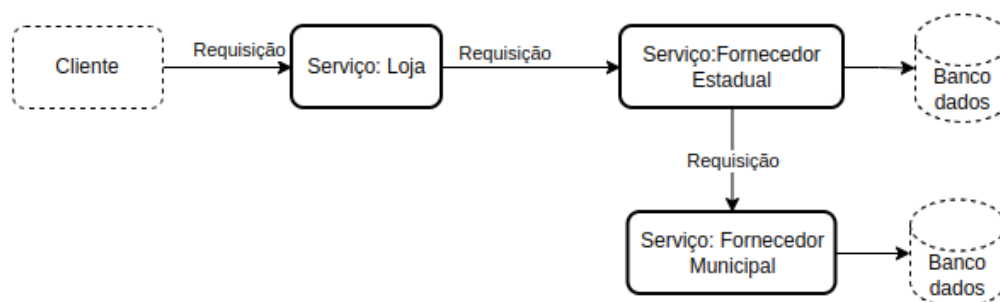
O projeto possui três microserviços que trocam informações entre si: Loja, Fornecedor Estadual e Fornecedor Municipal, formando um sistema com uma regra de negócio que simula a compra de itens por um cliente.

4.2.1.1 Serviços e Regra de Negócio

O planejamento visa criar um sistema de microserviços com uma regra de negócio para servir ao propósito de realizar o estudo da comunicação entre os serviços e as falhas que podem ocorrer, sejam pela comunicação ou por falhas na regra de negócio. Quando um desses serviços falha, o tratamento de exceções irá reportar à ferramenta proposta quem irá informar em qual serviço ocorreu a falha bem como o nome da classe, nome do método e por fim, em qual linha foi disparada a exceção inicial.

Os serviços, como exibido na Figura 8, formam um sistema que possui uma regra de negócio baseada em venda de produtos. Essa regra consiste em um modelo onde o cliente interage com o serviço Loja para efetuar um pedido de compra. Assim, ele informa os itens desejados e também o endereço de entrega. Na sequência, o serviço Loja comunica-se com o serviço Fornecedor Estadual, que consulta e verifica se há um fornecedor ao nível de estado e então comunica-se com o serviço Fornecedor Municipal, enviando a cidade, e aguarda uma resposta desse serviço. O serviço Fornecedor Municipal, então devolve o nome do fornecedor ao nível de município que pode atender à cidade que o cliente deseja entregar os itens comprados. O serviço Fornecedor Estadual retorna para o Serviço Loja os nomes dos Fornecedores que irão fazer parte da cadeia de entrega. O serviço Loja, finalizando a requisição, entrega a resposta ao cliente com os nomes de todos os fornecedores, estadual e municipal.

Figura 8 – Serviços Loja, Fornecedor Estadual e Fornecedor Municipal



Fonte: Elaborada pelo autor.

4.2.1.2 Número de Porta

Cada serviço precisa ter seu número de porta publicado no arquivo de configuração do projeto e fazem parte das URLs de requisições a um determinado serviço. O esquema de portas está exibido pela Tabela 1.

Serviço	Código	Porta
Loja	1	8080
Fornecedor Estadual	2	8081
Fornecedor Municipal	3	8082

Tabela 1 – Informações que identificam um serviço. Fonte: Elaborada pelo autor.

4.2.1.3 Classes de Exceções Adotadas para Testes

É possível fazer simulações de um cliente a solicitar uma compra e enviar dados incorretos, como um nome de estado e cidade errados e isso irá ocasionar falhas nos serviços que tratam essas informações. Esse cenário com regras de negócio será utilizado na fase de validação da ferramenta que deverá informar corretamente os erros que ocorreram e os escopos da falha.

Ao nível de teste, foi criada uma regra de negócio própria para simular exceções. Para tanto, define-se um grupo de exceções mapeadas a um código erro HTTP como exibido na Tabela 2. O código de erro HTTP é importante para encapsular o erro interno e apresentar ao cliente somente o número do erro. Assim,

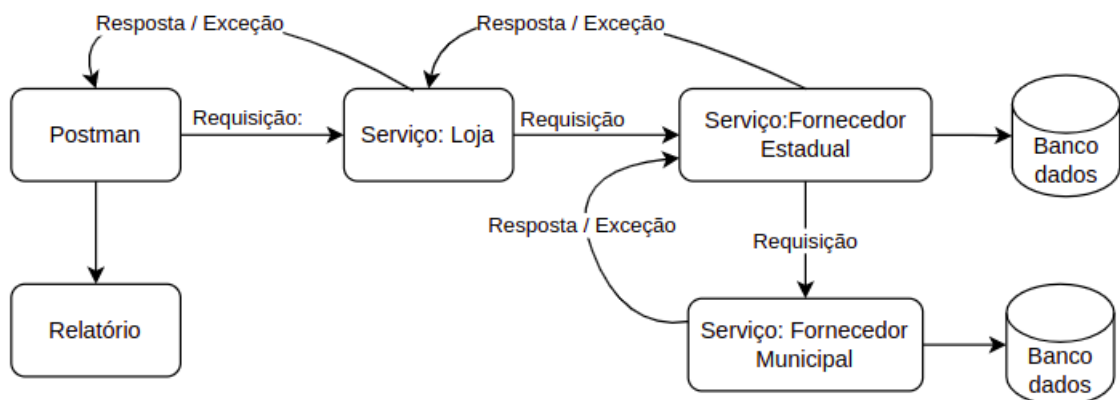
cada serviço possui uma classe⁶ que responde a uma solicitação e poderá lançar as exceções da Tabela 2, conforme a regra da requisição recebida do cliente. Na requisição chega o número do código da exceção que irá ser disparada e assim, o serviço valida essa requisição e pode direcionar o fluxo da execução para fazer uma divisão por zero, consultar a posição incorreta de um *array* e talvez apontar para uma posição na memória que não exista. Portanto, essas exceções citadas são disparadas em um escopo controlado e previsto.

Código	Classe - Causa Raiz	erro HTTP
1	ArithmeticException	500
2	ArrayIndexOutOfBoundsException	500
3	NullPointerException	404

Tabela 2 – Grupo de exceções utilizadas para serem causa raiz na fase de testes.
Fonte: Elaborada pelo autor.

A ferramenta Postman será utilizada para simular requisições de cliente ao sistema de compras, compostos pelos microserviços. Como exhibe Figura 9.

Figura 9 – Cenário de teste por simulação de requisição de cliente via Postman



Fonte: Elaborada pelo autor.

⁶ Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/service/ExceptionGenerator.java>

4.2.2 ExceptionTracker

A ferramenta proposta foi batizada de “ExceptionTracker”. Ela possui uma API *back-end* que irá dispor os *endpoints* para a *interface* gráfica, *front-end* da aplicação.

4.2.2.1 Back-end ExceptionTracker

Uma API desenvolvida em Node.js que conecta-se ao MongoDB e disponibiliza *endpoints* à *interface* gráfica.

4.2.2.2 Front-end ExceptionTracker

Uma API desenvolvida em VUE.js que se conecta ao *back-end* e obtém dados para geração dos relatórios para o usuário final.

4.3 Implementação

Esta etapa implementa o projeto com as características técnicas essenciais para que a ferramenta proposta funcione adequadamente: *traceId*, exceções encadeadas, mapeamento das falhas em códigos de erro HTTP e registro da pilha de exceções no MongoDB. Maiores detalhes serão exibidos Capítulo 5.

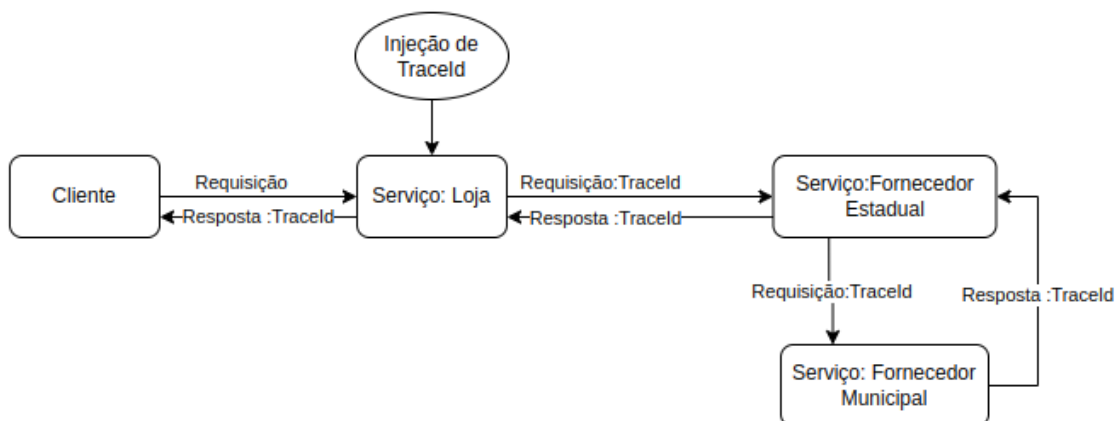
4.3.1 Microserviços

Implementação dos microserviços.

4.3.1.1 Injetar Identificador Único da Transação — *TraceID*

Uma requisição iniciada por um cliente deve terminar nesse cliente, com sucesso ou erro. Para garantir a rastreabilidade da transação que percorre vários serviços é necessário injeção de *traceId*, segundo exhibe a Figura 10. Uma vez registrado, o *traceID* pode ser obtido por qualquer serviço que processa a requisição no momento.

Figura 10 – Injeção de traceId na requisição.



Fonte: Elaborada pelo autor.

Implementar tratamento de exceções encadeadas e Mapear exceções em código de erros (HTTP)

Os serviços deverão ter tratamentos de exceções com encadeamento. O encadeamento evita perder as informações da pilha da exceção original ao ser lançada de um escopo a outro.

Em cada declaração *try...catch*, como exibido na 11, a exceção inicial é encapsulada na nova exceção que será lançada. A exceção é convertida para um código de erro HTTP para ser exibida no retorno da requisição como indica Figura 12.

4.3.1.2 Registrar Pilha de Exceções no MongoDB

A pilha de exceção é tratada para ser disponibilizada em formato JSON e então é salva no MongoDB Atlas, conforme a Figura 12.

A Figura 12 apresenta a arquitetura interna que um serviço deve conter para atender o propósito de se ter as exceções salvas corretamente no banco de dados. A parte mais importante é o bloco *exception handler*. O bloco *controller* filtra as requisições *HTTP* conforme os seus verbos. O bloco de *service* executa a regra de negócio solicitada pela requisição.

Figura 11 – Bloco try...catch: exceção encadeada.

```
@Service
public class ExceptionGenerator {
    public void arithmeticExceptionInitCauseGenerator() {
        try {
            int i = 4 / 0;
        } catch (ArithmeticException exception) {

            IOException ioException = new IOException("Falha
                ao obter informacoes do arquivo");
            ioException.initCause(exception);

            ServerErrorException serverErrorException = new
                ServerErrorException("Falha Interna",
                    ioException);
            throw serverErrorException;
        }
    }
}
```

Fonte: Elaborada pelo autor.

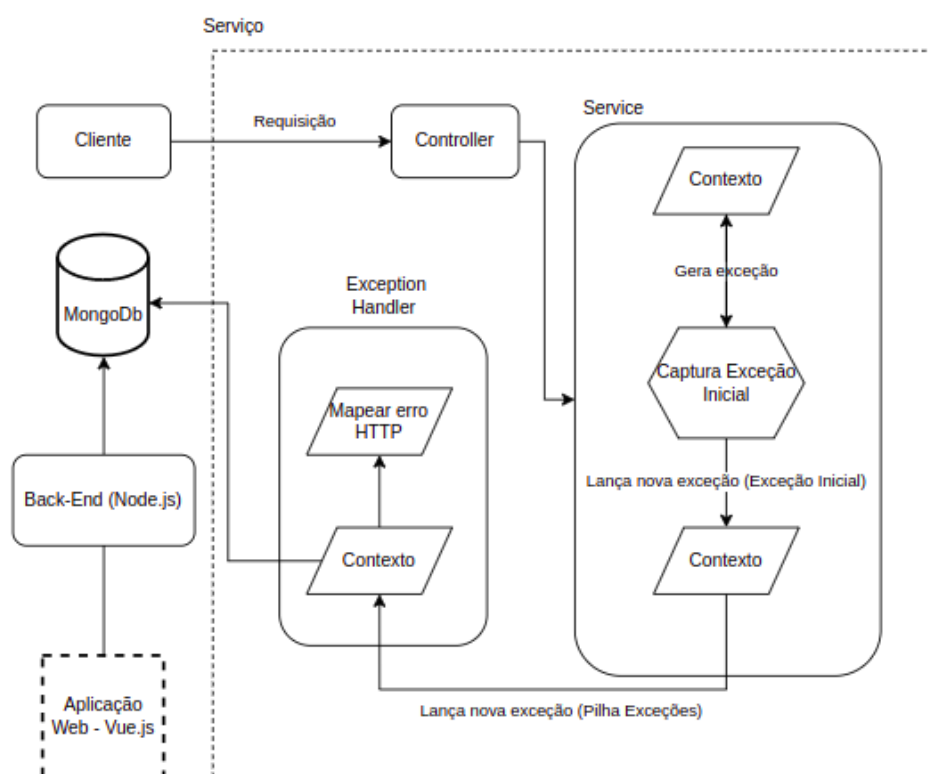
No Spring Boot é possível anotar uma classe para ser responsável por coletar todas as exceções que vierem ocorrer em todo o programa (*@RestControllerAdvice*) sendo referenciado no texto como *Exception Handler*.

4.3.2 ExceptionTracker

A API *back-end* é exibida conectada ao MongoDB e à aplicação gráfica, como exibido na Figura 12.

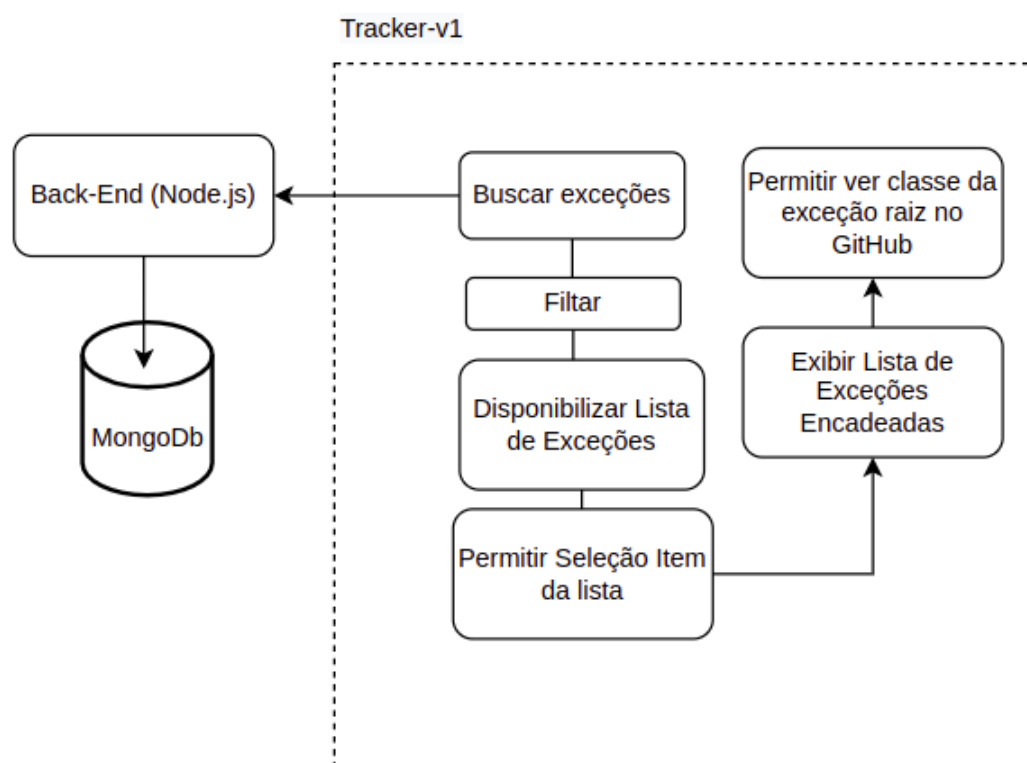
A ferramenta possui a parte gráfica para apoiar o desenvolvedor a encontrar as exceções que os microsserviços registraram no MongoDB, como exibido na Figura 13.

Figura 12 – Pilha de exceção é tratada e enviada ao MongoDB Atlas.



Fonte: Elaborada pelo autor.

Figura 13 – Aplicação para resolver e gerar informações de falhas no sistema.



Fonte: Elaborada pelo autor.

5 Desenvolvimento

O projeto completo pode ser acessado no GitHub¹.

A arquitetura ilustrada na Figura 12 foi esboçada para o propósito desse trabalho de gerenciar as exceções ocorridas nas requisições entre microserviços, e foi implementada sem maiores modificações. O escopo da falha é essencial e, portanto, os dados da classe, do método, da linha onde a exceção inicial foi disparada devem ser registrados corretamente no modelo de exceção a ser salvo no banco de dados. Diferente de outros trabalhos relacionados, não há necessidade de efetuar *logs* específicos para a exceção, a não ser fazer o tratamento de exceção exatamente como é abordado na literatura de programação orientada à objetos Java. Cada serviço é responsável por registrar no MongoDB as exceções, e então um outro serviço, o *back-end* da *interface* gráfica utiliza o mesmo banco de dados para fornecer *endpoints* para a *interface* gráfica realizar os tratamentos e exibir ao usuário dados organizados acerca das exceções que o sistema registrou.

Para o tratamento das exceções foi idealizado o *back-end* e *front-end* totalmente dedicados para consumir esses registros e processá-los para haver uma visualização da falha e seu escopo a partir de uma exceção escolhida pelo usuário para ser analisada, em uma lista disponibilizada, contendo todas as exceções registradas no banco de dados. A Tabela 3 ilustra a organização dessas definições:

Componente	Responsabilidade
Serviço	Arquitetura dos serviços.
MongoDb	Camada de persistência para exceções.
Back-End (ExceptionTracker)	Camada de interface entre o MongoDB e o front-end.
Front-End (ExceptionTracker)	Permite visualização gráfica e organizada das exceções.

Tabela 3 – Componentes da arquitetura e responsabilidades.

¹ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master>>.

5.1 Microserviços

Para a abordagem desse trabalho são utilizados três microserviços pequenos e responsáveis pelas tarefas definidas na Tabela 4 sendo exibidos na Figura 8.

Serviço	Responsabilidade
Loja	Principal, venda de itens de fornecedores estaduais.
Fornecedor Estadual	Fornece à loja itens de fornecedores das cidades.
Fornecedor Municipal	Distribui itens para o fornecedor Estadual.

Tabela 4 – Serviços e responsabilidades.

Esses serviços foram implementados utilizando a linguagem Java e o *framework Spring Boot*. Além disso, foram utilizadas diversas bibliotecas para permitir comunicação com banco de dados, comunicação síncrona entre serviços, injeção de identificador único na transação (traceId), diminuição de verbosidade e validações de entradas de usuários. Essas bibliotecas foram detalhadas na Seção 5.1.2.

A seguir, abordaremos também o ambiente de desenvolvimento e a arquitetura utilizados na implementação.

5.1.1 Ambiente de Desenvolvimento

O ambiente de desenvolvimento foi configurado conforme a Tabela 5 indica.

Item	Especificação
Computador IDEA	8 GIB RAM, Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz x 4.1.3
IntelliJ IDEA	IntelliJ IDEA 2021.1.3 (Community Edition)
Spring Boot	versão 2.5.6
MariaDB	Ver 15.1 Distrib 10.5.12-MariaDB
Postman	v9.13.1

Tabela 5 – Configuração ambiente de desenvolvimento.

5.1.2 Bibliotecas Utilizadas

Seis bibliotecas foram utilizadas como dependências para atender as características propostas para a ferramenta, discutidas na Seção 4.1.

- **Spring Cloud Sleuth:** possibilita injetar um identificador único (traceID) na transação que percorre vários microserviços.²
- **Spring Cloud Open Feign:** cliente HTTP usado para fazer chamadas síncronas entre APIs expostas pelos microserviços.³
- **Spring cloud:** providencia ferramentas para construção de padrões comuns em sistemas distribuídos.⁴

Lombok: auxilia a diminuir a verbosidade da linguagem Java por meio de anotações. Lombok encapsula propriedades como *getter* e *setter*.⁵

Starter-validation: auxiliar validação de entradas de usuários, via anotações em código.⁶

Mongodb-driver-sync: *driver* síncrono Java para permitir a integração Java e MongoDB.⁷

5.1.3 Arquitetura

Os serviços⁸ possuem basicamente a mesma estrutura de pacotes como indicado na Figura 14. A classe principal *MicroservicesSpringCloudApplication* possui o método *public static void main()* que inicializa a aplicação Spring, e possui a definição da (URL) de autenticação com o MongoDB .

Será realizado, a seguir, o detalhamento de cada pacote, com suas classes e responsabilidades, com base na estrutura do serviço *Fornecedor Estadual*⁹

² Disponível em: <<https://spring.io/projects/spring-cloud-sleuth>> Acesso em: 6 fev. 2022.

³ Disponível em: <<https://spring.io/projects/spring-cloud-openfeign>> Acesso em: 6 fev. 2022.

⁴ Disponível em: <<https://spring.io/projects/spring-cloud>> Acesso em: 6 fev. 2022.

⁵ Disponível em: <<https://www.baeldung.com/intro-to-project-lombok>> Acesso em: 6 fev. 2022.

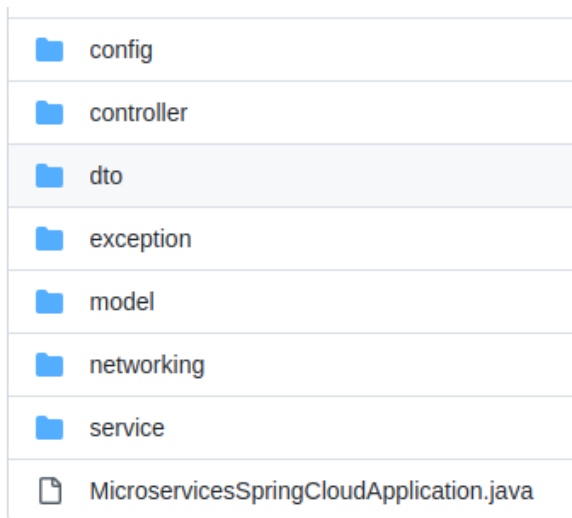
⁶ Disponível em: <<https://www.baeldung.com/spring-boot-bean-validation>> Acesso em: 6 fev. 2022.

⁷ Disponível em: <<https://mongodb.github.io/mongo-java-driver/3.8/driver/getting-started/installation>>

⁸ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud>>

⁹ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado>>

Figura 14 – Modelagem de pacotes Java

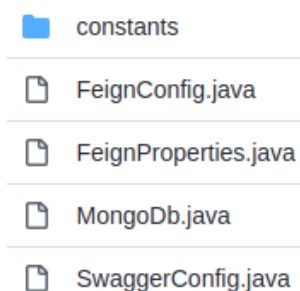


Fonte: Elaborada pelo autor.

5.1.3.1 Config

O pacote **Config**¹⁰ possui a responsabilidade de agrupar classes que têm por objetivo a configuração das bibliotecas utilizadas no projeto e do próprio projeto.

Figura 15 – Pacote config



Fonte: Elaborada pelo autor.

A Figura 15 ilustra o conteúdo do pacote Config. As classes configuram as

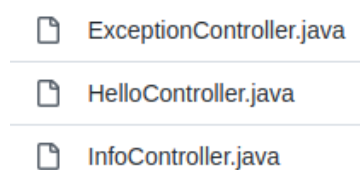
¹⁰ Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/config>

bibliotecas utilizadas, definidas na Seção 5.1.2.

5.1.3.2 Controller

As classes no pacote **Controller**¹¹ são responsáveis pelo processamento das requisições API REST, preparando o modelo e retornando a resposta da requisição. São anotadas com a anotação *@RestController* para indicar classes que gerenciam requisições.¹²

Figura 16 – Pacote Controller



Fonte: Elaborada pelo autor.

5.1.3.3 Service

As classes no pacote **Service**¹³ são responsáveis pela lógica do modelo e para tal recebem a anotação *@Service*.¹⁴

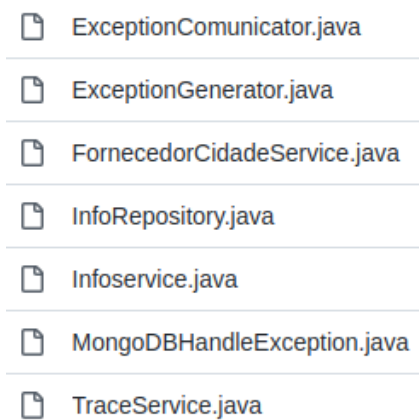
¹¹ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/controller>>

¹² Disponível em: <<https://stackabuse.com/controller-and-restcontroller-annotations-in-spring-boot/>> Acesso em: 6 jun. 2022.

¹³ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/service>>

¹⁴ Disponível em: <https://www.tutorialspoint.com/spring_boot/spring_boot_service_components.htm> Acesso em: 6 jun. 2022.

Figura 17 – Pacote Service



Fonte: Elaborada pelo autor.

5.1.3.4 Dto

As classes no pacote **Dto**¹⁵ são responsáveis por estruturar e definir quais os dados do modelo podem ser transferidos em respostas de requisições.

Figura 18 – Pacote Dto



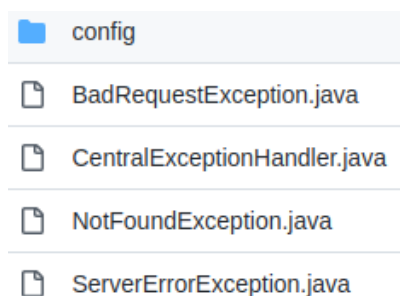
Fonte: Elaborada pelo autor.

¹⁵ Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/dto>

5.1.3.5 Exception

As classes no pacote **Exception**¹⁶ são classes que definem exceções customizadas para o projeto.¹⁷

Figura 19 – Pacote Exception

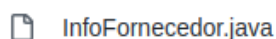


Fonte: Elaborada pelo autor.

5.1.3.6 Model

As classes no pacote **Model**¹⁸ são responsáveis pela persistência no banco de dados *MariaDB*, utilizado para armazenar informações do negócio, e para tal recebem as anotações `@Entity` e `@Table`.¹⁹

Figura 20 – Pacote Model



Fonte: Elaborada pelo autor.

¹⁶ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/exception>>

¹⁷ Disponível em: <<https://www.baeldung.com/java-new-custom-exception>> Acesso em: 6 jun. 2022.

¹⁸ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/model>>.

¹⁹ Disponível em: <<https://www.baeldung.com/jpa-entity-table-names>> Acesso em: 6 jun. 2022.

5.1.3.7 NetWorking

As classes no pacote **NetWorking**²⁰ são classes que definem o processo de comunicação entre serviços. O sub-pacote *config* possui classes de configurações de erros e respostas de requisições e o sub-pacote *config* possui métodos que permitem a comunicação com outros serviços, logo, possui informações de rotas para requisições aos serviços com quais o projeto foi configurado para comunicar-se.

Figura 21 – Pacote NetWorking



Fonte: Elaborada pelo autor.

5.1.3.8 Configuração da Aplicação

Em cada projeto de serviço existe o pacote **resources**²¹ que possui um arquivo que configura a aplicação por completo: **application.properties**.

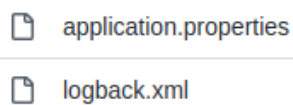
Como ilustrado na Figura 23 nesse arquivo configura-se a porta onde a aplicação será publicada no servidor. Indica os *endpoints*,²² sendo uma *interface* disponibilizada por outro serviço ao qual se deseja utilizar os recursos. Além disso, o arquivo possui as configurações com o banco de dados MariaDB.

²⁰ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/networking>>.

²¹ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/resources>>

²² Disponível em: <<https://codejagd.com/pt/diferenca-endpoint-e-api>> Acesso em: 6 jun. 2022.

Figura 22 – Application.properties



Fonte: Elaborada pelo autor.

Figura 23 – Application.properties: Configura a aplicação.

```
server.port=8081
server.servlet.context-path=/fornecedorEstado
service.name=fornecedorEstado
service.base.package=alura
service.path=https://github.com/aritana/tcc2-exceptionTracker/blob/master\
/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java

#Feign time out configure
feign.client.config.default.connectTimeout=28000
feign.client.config.default.readTimeout=28000
#feign.client.config.default.loggerLevel=basic

#Endpoints
endpoint.fornecedor.url=http://localhost:8082/fornecedorCidade

#https://spring.io/guides/gs/accessing-data-mysql/
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/test_db
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name =org.mariadb.jdbc.Driver
```

Fonte: Elaborada pelo autor.

5.1.3.9 Rotas

- **Get *Exception:exception/service***: gera uma exceção causa raiz com código conforme Tabela 2 no serviço com código respectivo à Tabela 1.

5.1.4 Principais Funcionalidades Implementadas

As principais funcionalidades implementadas na Seção 5.1, que trata dos microsserviços utilizados neste estudo, foram:

- **Injetar Identificador Único da transação - TraceID**: como mencionado na Seção 2.4, a instalação da biblioteca Spring Cloud Sleuth, conforme pode ser visualizado no arquivo de dependências *pom.xml*²³, injeta um identificador único nas requisições HTTP, como ilustrado na Figura 24, que representa a resposta de uma requisição que o usuário recebeu ao ocorrer uma falha de comunicação entre dois serviços.

Este ID único, aqui denominado, *traceId*, pode ser obtido facilmente em qualquer serviço, por meio da classe *TraceService.java* disponível em todos os serviços.²⁴

- **Exceções Encadeadas**: Exceções encadeadas permitem relacionar exceções. Com essa técnica, foi possível mapear a causa raiz de exceções para este estudo.²⁵
- **Centralizar Captura de Exceções**: No framework Spring Boot é possível anotar uma classe com a anotação *@RestControllerAdvice*, para que essa classe seja responsável por coletar todas as exceções, que o desenvolvedor definir, que vierem a ocorrer no programa todo, antes de ser enviada no retorno da requisição. Nessa camada, as exceções são mapeadas para códigos

²³ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/pom.xml>>

²⁴ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/service/TraceService.java>>

²⁵ Disponível em: <<https://www.geeksforgeeks.org/chained-exceptions-java/>> Acesso em: 6 fev. 2022.

Figura 24 – TraceID: ID único da requisição/resposta HTTP

```
1  {
2      "timestamp": "19:37:01.320450459",
3      "status": "500",
4      "error": "Internal Server Error",
5      "message": "Falha Interna",
6      "trace_id": "1704267e79bdab09"
7  }
```

Fonte: Elaborada pelo autor.

de respostas HTTP, sendo tratadas para serem exibidas em um formato definido no projeto. A Figura 24 exibe o formato definido neste projeto para exibição de exceções como respostas de requisições HTTP e a Figura 25 exibe um trecho do código da classe *CentralExceptionHandler*.²⁶

- **Persistir Exceções no MongoDB:** classe *CentralExceptionHandler* centraliza as exceções e então consome métodos da classe de serviço *MongoDBHandleException* que possui a lógica para salvar a pilha de exceção no MongoDB.²⁷ Cada serviço consegue acessar a mesma aplicação do MongoDB, sendo um serviço baseado em nuvem, com versão gratuita limitada a 512 MB de armazenamento.

²⁶ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/exception/CentralExceptionHandler.java>>

²⁷ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/service/MongoDBHandleException.java>>.

Figura 25 – Bloco Exception Handler: Onde todas as exceções são observadas.

```
@RestControllerAdvice
public class CentralExceptionHandler {

    @Autowired
    TraceService traceService;
    @Autowired
    MongoDBHandleException mongoDBHandleException;

    private static Logger logger =
        LoggerFactory.getLogger(SLF4JLogger.class);

    @ResponseStatus(code = HttpStatus.NOT_FOUND)
    @ExceptionHandler(NotFoundException.class)
    public ResponseError handleNotFound(NotFoundException exception) {
        ResponseError responseError;
        if (exception.getResponseError() == null) {
            responseError = ResponseError.builder()
                .timestamp(String.valueOf(LocalTime.now()))
                .status("404")
                .error(HttpStatus.NOT_FOUND.getReasonPhrase())
                .trace_id(traceService.getTraceId())
                .message(exception.getMessage()).build();
        }
    }
}
```

Fonte: Elaborada pelo autor.

- **Geração de exceções em um serviço:** com a ferramenta Postman²⁸ é possível simular requisições que geram exceções como causa raiz, conforme Tabela 2, nos serviços desejados, como Tabela 1, e com isso é possível validar se o *front-end* rastreia a causa raiz corretamente.

Para exemplificar, considere a seguinte situação: o cliente, neste estudo de caso, para gerar uma exceção no serviço “Fornecedor Municipal”, cujo código é 3, conforme Tabela 1; e deseja neste serviço provocar uma falha (e.g divisão por zero) que gere uma exceção "NullPointerException", cujo código é 3, conforme Tabela 2, deve prosseguir com a seguinte URL:

`http://localhost:{numeroPortaServico}/fornecedorCidade/exception/{codigoExcecao}/{codigoServico}` conforme rota descrito na Seção 5.2.3.

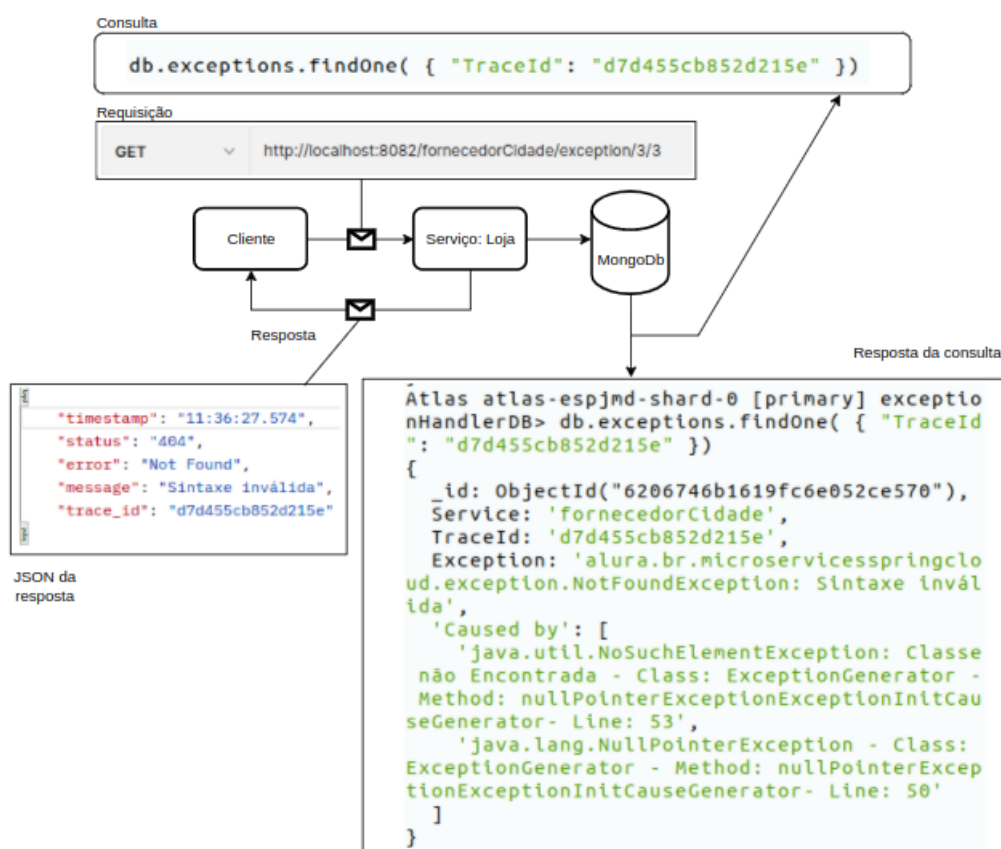
que adaptada, conforme códigos nos campos `codigoExcecao` e `codigoServico` fica: `http://localhost:8080/fornecedorCidade/exception/3/3`.

O cliente só se comunica com o serviço “Loja”, cuja porta é 8080, conforme ilustra a Figura 8 e Tabela 1.

Dessa forma, o resultado para essa solicitação pode ser visualizado na Figura 26.

²⁸ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Postman>>

Figura 26 – Resultado para uma requisição de teste

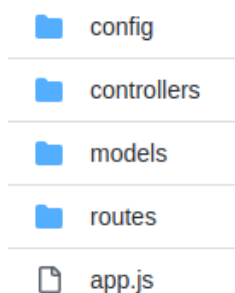


Fonte: Elaborada pelo autor.

5.2 Back-end ExceptionTracker

A API TrackerCentral²⁹ desenvolvida em Node.js, conecta-se ao MongoDB e disponibiliza *endpoints* à *interface* gráfica, que gerará os relatórios gráficos para o usuário final.

Figura 27 – TrackerCentral



Fonte: Elaborada pelo autor.

5.2.1 Conexão MongoDB

Para conectar ao banco MongoDB, utilizou-se a biblioteca Mongoose³⁰, configurada no arquivo *dbConnect*.³¹

5.2.2 Schema de persistência no banco de dados

O arquivo *Exception.js*³², disponível no pacote *Models* configura o modelo esquemático (*Schema*), um objeto JSON, que define a estrutura e o conteúdo dos dados.³³ O modelo esquemático pode ser visualizado pela Figura 28, obtido pela ferramenta Postman.

²⁹ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/trackerCentral>>

³⁰ Disponível em: <<https://mongoosejs.com/docs/index.html>> Acesso em: 6 fev. 2022.

³¹ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/trackerCentral/src/config/dbConnect.js>>

³² Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/trackerCentral/src/models/Exceptions.js>>

³³ Disponível em: <<https://mongoosejs.com/docs/guide.html>> Acesso em: 6 fev. 2022.

Figura 28 – *Schema* do objeto Exceção

```

{
  "_id": "6296e026151d3c3f160cae06",
  "service": "fornecedorCidade",
  "traceId": "49dd03f55d992a92",
  "exception": "alura.br.microservicespringcloud.exception.ServerErrorException: Falha Interna",
  "causedBy": [
    "java.io.IOException: Falha ao obter informações do arquivo - Class: ExceptionGenerator - Method: arithmeticExceptionInitCauseGenerator - Line: 22",
    "java.lang.ArithmeticException: / by zero - Class: ExceptionGenerator - Method: arithmeticExceptionInitCauseGenerator - Line: 18"
  ],
  "path": "https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicespringcloud/service/ExceptionGenerator.java"
}

```

Fonte: Elaborada pelo autor.

5.2.3 Rotas

- ***Get Exception por ID***: realiza consulta pelo ID de uma exceção no banco de dados MongoDB e retorna a exceção encontrada ou *null*.
- ***Get Exception por TraceId***: realiza consulta pelo TraceId de uma exceção no banco dados MongoDB e retorna a exceção encontrada ou *null*.
- ***Get Exceptions***: realiza consulta pelo de todas exceções presentes no MongoDB e retorna uma lista de exceções ou uma lista vazia.
- ***Delete Exception por ID***: realiza consulta pelo ID de uma exceção no banco de dados MongoDB e deleta a exceção encontrada ou devolve mensagem de erro caso não a encontre.
- ***Delete Exception por TraceId***: realiza consulta pelo TraceId de uma exceção no banco de dados MongoDB e deleta a exceção encontrada ou devolve mensagem de erro caso não a encontre.

5.2.4 Principais Funcionalidades Implementadas

As principais funcionalidades implementadas na Seção 5.2 se resumem na disponibilização, pelo *back-end*, de uma API com os *endpoints* necessários para o funcionamento correto do *front-end*. Todavia, apenas duas das rotas disponibilizadas

pelo *back-end*: *Get Exceptions* e *Delete Exception por TraceId* serão consumidas, as restantes poderão ser utilizadas em trabalhos futuros.

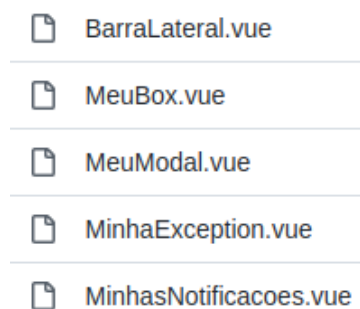
5.3 Front-end ExceptionTracker

A API ExceptionTracker³⁴ foi desenvolvida em *VUE.js*, um *framework* para criação de *interfaces*³⁵. A API se conecta ao *back-end* trackerCentral para disponibilizar a *interface* gráfica ao usuário final.

5.3.1 Componentes

Componentes são instâncias reutilizáveis do Vue.³⁶ Os componentes necessários para criação do esquema visual estão ilustrados na Figura 29. O componente mais importante, MinhaException.vue, representa uma exceção no topo da pilha de exceções encadeadas obtidas do *back-end*.

Figura 29 – TrackerCentral



Fonte: Elaborada pelo autor.

³⁴ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Front/tracker-v1>>

³⁵ Disponível em: <<https://vuejs.org/>> Acesso em: 6 fev. 2022.

³⁶ Disponível em: <<https://br.vuejs.org/v2/guide/components.html>> Acesso em: 6 fev. 2022.

5.3.2 Rotas

As rotas, descritas no arquivo `router/index.ts`³⁷, definem as rotas da aplicação e permitem a transição entre *views* conforme interação com o usuário.

5.3.3 Views

As *views*³⁸ permitem criar visualizações a partir de arranjo de componentes sendo roteáveis³⁹. A Figura 30 ilustra as *views* do projeto.

Figura 30 – TrackerCentral



Fonte: Elaborada pelo autor.

³⁷ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Front/tracker-v1/src/router/index.ts>>

³⁸ Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Front/tracker-v1/src/views>>

³⁹ Disponível em: <<https://router.vuejs.org/guide/essentials/named-views.html>> Acesso em: 6 fev. 2022.

5.3.4 Principais Funcionalidades Implementadas

As principais funcionalidades implementadas na Seção 5.3, conforme exibido na Figura 13, são descritos a seguir:

- **Buscar Exceções:** As exceções são obtidas pela API `trackerCentral`. A execução em ambiente local, utilizado navegador Chrome e necessitou da extensão *Allow CORS*⁴⁰ para permitir o acesso de dados entre as aplicações.
- **Filtrar:** Toda exceção encadeada tem no mínimo dois níveis, pois como definido na Seção 5.1.4, cada serviço possui uma camada *Exception Handler* que faz o mapeamento das exceções internas para um modelo de resposta mapeado para o protocolo HTTP. Essa exceção mais externa é declarada no modelo de objeto *JSON* como **exception** e a pilha de exceções é declarada no mesmo modelo como **causedBy**, um *array* com todas exceções encadeadas inerentes ao código do projeto, como pode ser observado na Figura 28.

Diante disso, ao utilizar o filtro que colete apenas registros com o campo **causedBy**, simplifica-se e obtêm-se as exceções com causa raiz e sem duplicidade.

- **Disponibilizar Lista de Exceções:** As exceções são listadas conforme Figura 31.
- **Permitir seleção de item da Lista de Exceções:** As exceções listadas são clicáveis, e o usuário pode selecionar alguma, conforme Figura 32.
- **Exibir lista de exceções encadeadas:** A exceção selecionada possui o encadeamento de outras exceções que devem ser exibidas, com a exceção raiz em destaque no fim da Tabela. Algo notável é que cada exceção possui informações do escopo da falha, o nome da classe, o nome do método, e a linha onde ocorreu a exceção, como ilustrado na Figura 33.
- **Permitir ver o escopo da exceção no GitHub:** Ao clicar no botão *Ver Classe*, ilustrado na Figura 33, a ferramenta direciona para o código no

⁴⁰ Disponível em: <<https://mybrowseraddon.com/access-control-allow-origin.html>> Acesso em: 6 fev. 2022.

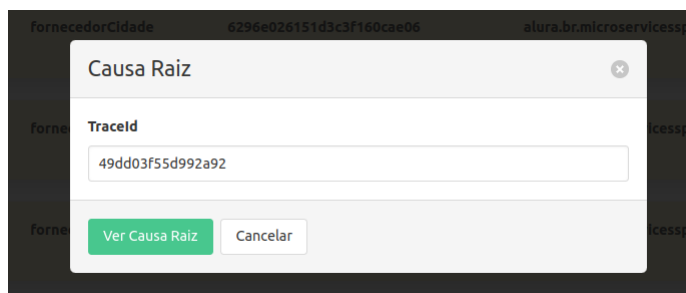
Figura 31 – ExceptionTracker: lista de exceções.



TraceId	Service	Id	Exception
49dd03f55d992a92	fornecedorCidade	6296e026151d3c3f160cae06	alura.br.microservicespringcloud.exception.ServerErrorException: Falha interna
2dbb5d2f2d710b69	fornecedorCidade	6296e042151d3c3f160cae08	alura.br.microservicespringcloud.exception.ServerErrorException: Falha interna
7e6c8d906ffbae1f	fornecedorEstado	6296e055031ef03f08493910	alura.br.microservicespringcloud.exception.ServerErrorException: Falha interna

Fonte: Elaborada pelo autor.

Figura 32 – ExceptionTracker: seleção de exceção para verificar sua causa raiz.



Fonte: Elaborada pelo autor.

Figura 33 – ExceptionTracker: exceções encadeadas com causa raiz em destaque.

excecoes

← Voltar
Ver Classe

Serviço: fornecedorEstado Chained Exceptions: A última exceção na tabela é a exceção raiz.	
java.lang.UnsupportedOperationException: Operacao não permitida - Class: ExceptionGenerator - Method: arrayIndexOutOfBoundsExceptionHandlerInitCauseGenerator - Line: 38	Caused By: ↓
java.lang.ArrayIndexOutOfBoundsException: Index 11 out of bounds for length 10 - Class: ExceptionGenerator - Method: arrayIndexOutOfBoundsExceptionHandlerInitCauseGenerator - Line: 35	Caused By: ↓

Delete Exception Register

Fonte: Elaborada pelo autor.

GitHub, após clicar no *link*, como exibido na Figura 34. O código exibido é respectivo à classe Java onde ocorreu a exceção, nos microsserviços. Com as informações de nome de método e número da linha, disponíveis, o usuário pode localizar o escopo.

Figura 34 – ExceptionTracker: link para acessar o escopo no GitHub.

excecoes

← Voltar

Link do Git Hub para exibição da classe de onde o erro foi propagado inicialmente:

<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/service/ExceptionGenerator.java>

Fonte: Elaborada pelo autor.

5.3.5 Informações Acerca do Projeto de Estudo de Caso

Para a realização desse trabalho, o autor realizou diversos cursos para buscar melhor embasamento para o projeto além da experiência prática obtida por meio de estágio não supervisionado. Os cursos realizados foram:

- **Microserviços Java:** os cursos relevantes para esse tema foram: *Spring Boot API REST: construa uma API*⁴¹ que concedeu conceitos para a criação de uma API Rest e *Microservices com Spring Cloud: Registry, Config Server e Distributed Tracing*⁴² que permitiu a compreensão da comunicação entre serviços e a injeção de identificador único nas requisições.
- **MongoDB e Node.js:** os cursos relevantes para esse tema foram: *MongoDB: uma alternativa aos bancos relacionais tradicionais*⁴³ que possibilitou a compreensão dos principais conceitos de MongoDB e *Node.js: API Rest com Express e MongoDB*⁴⁴ que ensinou a construir uma API de *back-end* em Node.js que utiliza o MongoDB como banco de dados.
- **Vue.js:** os cursos relevantes para esse tema foram: *Formação VUE.js*⁴⁵ que possibilitou a compreensão de todos os aspectos do *framework* para a construção da *interface* gráfica.

⁴¹ Disponível em: <<https://cursos.alura.com.br/course/spring-boot-api-rest>> Acesso em: 6 fev. 2022.

⁴² Disponível em: <<https://cursos.alura.com.br/course/microservices-spring-cloud-service-registry-config-server>> Acesso em: 6 fev. 2022.

⁴³ Disponível em: <<https://cursos.alura.com.br/course/mongodb>> Acesso em: 6 fev. 2022.

⁴⁴ Disponível em: <<https://cursos.alura.com.br/course/nodejs-api-rest-express-mongodb>> Acesso em: 6 fev. 2022.

⁴⁵ Disponível em: <<https://cursos.alura.com.br/formacao-vuejs3>> Acesso em: 6 fev. 2022.

6 Avaliação

A avaliação foi gravada e está disponível no Youtube para auxiliar no entendimento¹.

A estrutura da avaliação é exibida na Figura 35, a título de exemplo, ilustra uma situação de falha que inicia quando uma requisição iniciada na ferramenta Postman, que atua como o cliente da transação. A requisição percorre o serviço Loja e então alcança o serviço Fornecedor Estadual. No Serviço Fornecedor Estadual ocorre uma exceção do tipo `ArithmeticException`, registrada em seguida no banco Mongo e então a requisição retorna ao cliente a resposta com a mensagem de erro e o número do `traceId`, que identifica a requisição. O desenvolvedor interage com a interface gráfica informando esse número. A interface gráfica exibe um relatório que contém a informação que revela em qual serviço, nome da classe, método e linha onde foi disparada a primeira exceção, a causa raiz. Além disso, um *link* para o código no GitHub é fornecido para que a visualização se torne imediata.

Para avaliação da ferramenta foi utilizado o seguinte cenário:

Falha na regra de negócio: quando uma informação é enviada incorretamente ocorre uma falha e a requisição não será bem sucedida.

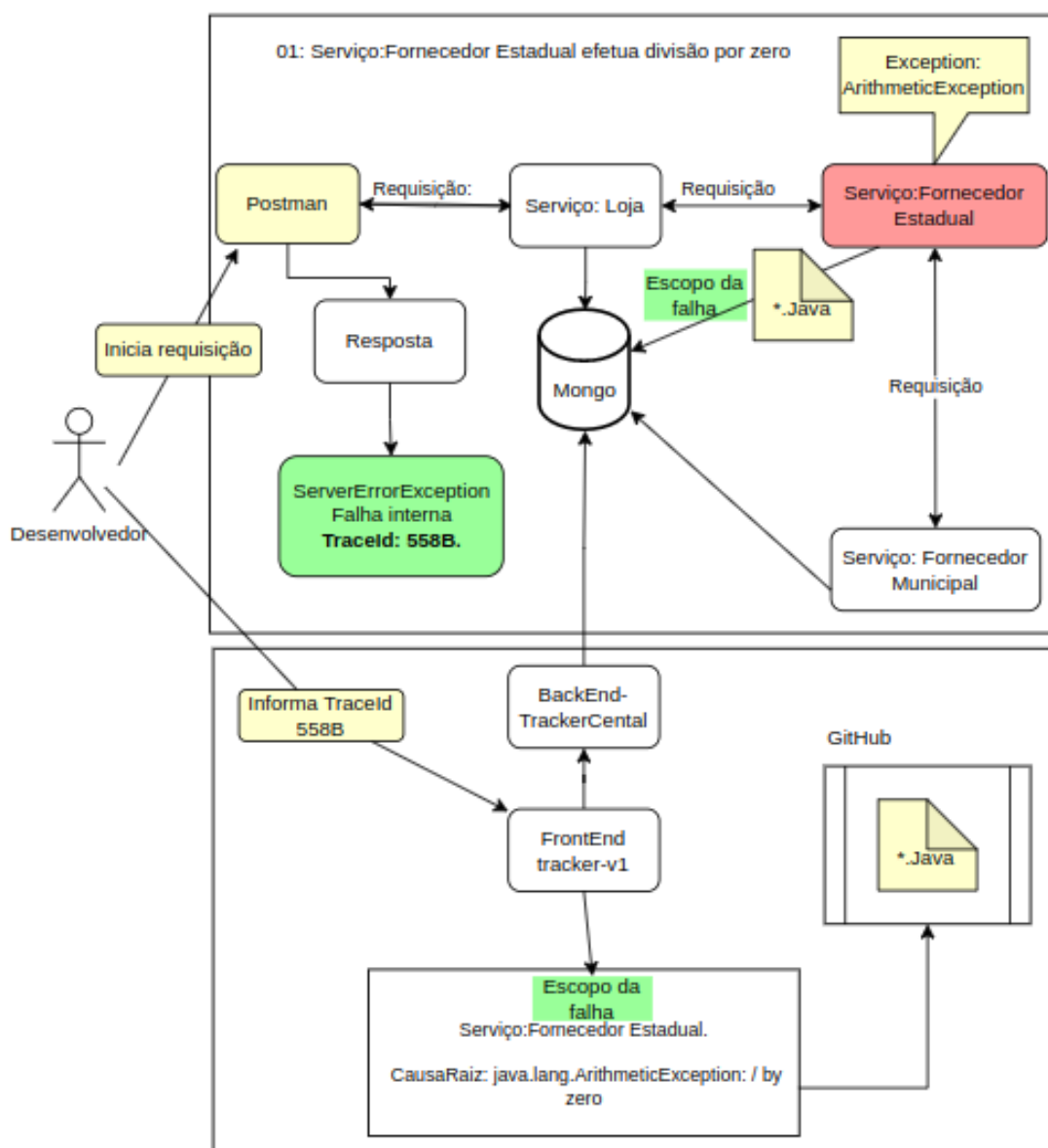
6.1 Falha na Regra de Negócio

Uma requisição com um erro de grafia no campo de endereço do estado é exibido na Figura 36. A grafia correta seria “Minas Gerais” e o campo possui o valor “Minas Cerais”.

O serviço que lida com informações de Fornecedores no nível de estado é o Fornecedor Estadual, exibido na Figura 8. Este serviço irá procurar por “Minas Cerais” e não irá encontrar no seu banco de dados e a requisição terá uma resposta de erro, contendo o `traceId` `df6680cdc5a20612`, como exibido na Figura 37.

¹ Disponível em: <<https://www.youtube.com/watch?v=BvkMEPnxAlM>>

Figura 35 – ExceptionTracker: avaliação de falha no serviço Fornecedor Estadual.

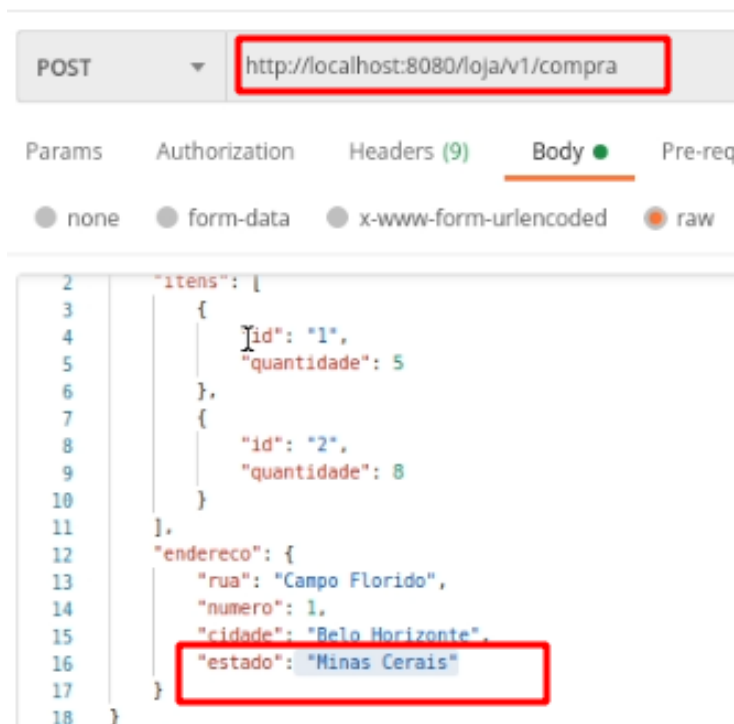


Fonte: Elaborada pelo autor.

O desenvolvedor irá consultar esse traceId na aplicação gráfica, como exibido na Figura 38.

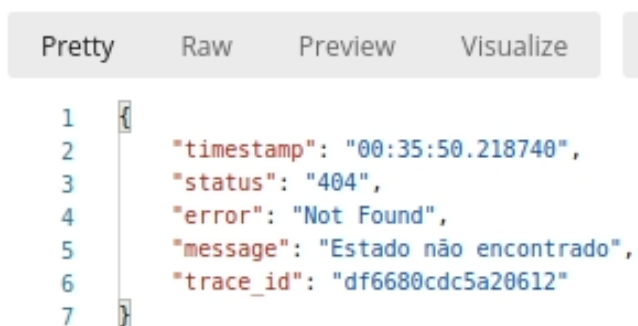
O usuário seleciona sendo direcionado à outra tela que fornece a informação

Figura 36 – ExceptionTracker: requisição com erro de grafia no campo de endereço do estado.



Fonte: Elaborada pelo autor.


Figura 37 – ExceptionTracker: erro de resposta da requisição.



Fonte: Elaborada pelo autor.

do escopo da falha, exibido na Figura 39:

Figura 38 – ExceptionTracker: busca pelo traceId df6680cdc5a20612.



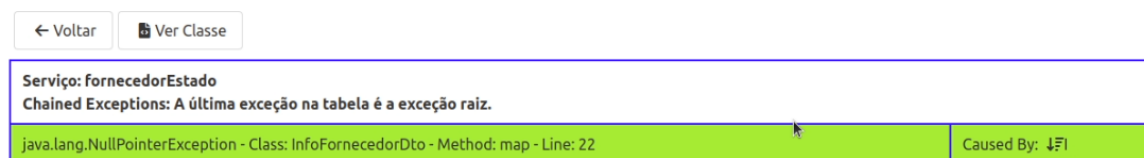
TraceId	Service	Id	Exception
df6680cdc5a20612	fornecedorEstado	62be6b966fa1b237e8b8cf43	alura.br.microservicesspringcloud.exception.NotFoundException: Estado não encontrado

Fonte: Elaborada pelo autor.

- **Serviço que falhou:** serviço Fornecedor Estado.
- **Exceção:** NullPointerException.
- **Classe:** InfoFornecedorDto.
- **Método:** map.
- **Linha:** linha 22.

Figura 39 – ExceptionTracker: escopo da falha.

excecoes



excecoes	
← Voltar	Ver Classe
Serviço: fornecedorEstado	
Chained Exceptions: A última exceção na tabela é a exceção raiz.	
java.lang.NullPointerException - Class: InfoFornecedorDto - Method: map - Line: 22	Caused By: ↓

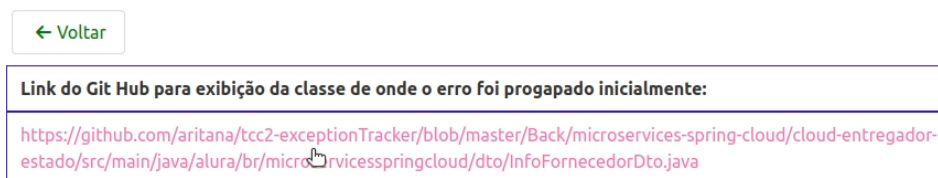
Fonte: Elaborada pelo autor.

O usuário interage com o botão “Ver Classe”, direcionado à tela que fornece um *link* para o código ser exibido no diretório do GitHub, exibido na Figura 40:

Em seguida, após a interação com o *link* a página do GitHub² é aberta para o usuário analisar o causa raiz como exibido na Figura 41.

² Disponível em: <<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/dto/InfoFornecedorDto.java>>

Figura 40 – ExceptionTracker: link do código no GitHub.



Fonte: Elaborada pelo autor.

Figura 41 – ExceptionTracker: código no GitHub.

```
19 public static InfoFornecedorDto maybe(InfoFornecedor infoFornecedor, InfoFornecedorCidadeDto infoFornecedorCidadeDto) throws NotFoundException {
20     InfoFornecedorDto infoFornecedorDto = null;
21     try {
22         String nomeFornecedorEstado = infoFornecedor.getNome();
23         String estado = infoFornecedor.getEstado();
24         String nomeFornecedorCidade = infoFornecedorCidadeDto.getNome();
25         String cidade = infoFornecedorCidadeDto.getCidade();
26
27         infoFornecedorDto = InfoFornecedorDto.builder()
28             .nomeFornecedorEstado(nomeFornecedorEstado)
29             .estado(estado)
30             .nomeFornecedorCidade(nomeFornecedorCidade)
31             .cidade(cidade)
32             .build();
33     } catch (NullPointerException exception) {
34         throw new NotFoundException("Estado não encontrado", exception);
35     }
36     return infoFornecedorDto;
37 }
38 }
```

Fonte: Elaborada pelo autor.

7 Limitações da Ferramenta

Nesse capítulo, são discutidas as principais limitações desse trabalho.

A qualidade final da atividade registrar adequadamente as exceções no MongoDB exige que seja feita o tratamento de exceções utilizando o conceito de exceções encadeadas como ilustra a Figura 42. Isso é necessário porque a prática de exceções encadeadas visa persistir o escopo inicial da falha mesmo se a exceção for lançada e capturada em outro escopo.

Figura 42 – Bloco try...catch: exceção encadeada.

```
@Service
public class ExceptionGenerator {
    public void arithmeticExceptionInitCauseGenerator() {
        try {
            int i = 4 / 0;
        } catch (ArithmeticException exception) {
            IOException ioException = new IOException("Falha ao obter
                informacoes do arquivo");
            ioException.initCause(exception);
            ServerErrorException serverErrorException = new
                ServerErrorException("Falha Interna", ioException);
            throw serverErrorException;
        }
    }
}
```

Fonte: Elaborada pelo autor.

As exceções lançadas, na última camada da pilha de exceções necessitam estar declaradas na classe `CentralExceptionHandler.java`¹ ou classe `CentralExceptionHandler` deve ter uma anotação para alguma classe de exceção mais genérica,

¹ Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/exception/CentralExceptionHandler.java> Acesso em: 6 fev. 2022.

na qual a exceção que foi lançada herda, como a classe de exceções `Exception` ou `RuntimeException`. Isso ocorre porque `CentralExceptionHandler` captura automaticamente todas exceções para qual foi configurada, quando essas exceções são lançadas e não são tratadas por nenhuma parte do código. Isso ocorre antes que a resposta da requisição é enviada de volta ao cliente. A Figura 43 ilustra a classe `CentralExceptionHandler` com seu método de captura de exceções do tipo `ServerErrorException`.

Figura 43 – `CentralExceptionHandler`: captura e salva a exceção no MongoDB.

```
@ResponseStatus(code = HttpStatus.INTERNAL_SERVER_ERROR)
@ExceptionHandler(ServerErrorException.class)
public ResponseError handleServerError(ServerErrorException
    exception) {
    ResponseError responseError;

    if (exception.getResponseError() == null) {
        responseError = ResponseError.builder()
            .timestamp(String.valueOf(LocalTime.now()))
            .status("500")
            .error(HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase())
            .trace_id(traceService.getTraceId())
            .message(exception.getMessage()).build();
    } else {
        responseError = exception.getResponseError();
    }
    logger.debug("ServerErrorException {}", exception.getMessage());
    mongoDBHandleException.saveException(exception);
    return responseError;
}
```

Fonte: Elaborada pelo autor.

A outra limitação está relacionada a necessidade de criação de uma classe, `MongoDBHandleException`, para adequar a exceção posterior e registro de dados no MongoDB. Isso é intrusivo, pois demanda inserção de classes no projeto.

Por fim, nesse trabalho não foi implementada a classificação das exceções

por níveis de criticidade para ser possível trazer ao usuário, pontos de observação imediata, algo que os trabalhos atuais carecem, segundo descrito na literatura.

8 Conclusões e Trabalhos Futuros

Ao lidar com uma requisição, algum erro pode ocorrer e uma instância de serviço pode lançar uma exceção. A exceção pode ser um sintoma de uma falha portanto deve ser verificada para que o sistema não fique comprometido. A arquitetura de Microserviços é distribuída e encontrar uma falha não é uma tarefa simples como na arquitetura Monolítica. Deve-se pesquisar os arquivos de *logs* de falhas, conceber julgamentos, fazer depuração, e reproduzir a falha.

A proposta desse trabalho foi apresentar uma ferramenta para monitoramento de exceções de software de maneira automatizada, com interface simples para permitir ao desenvolvedor que obtenha as informações das falhas com maior agilidade e evitar usar seu tempo para consultas de arquivos de arquivos de *logs* e sem necessidade de maturidade técnica que as ferramentas atuais exigem aos desenvolvedores que as utilizam. A ferramenta cumpriu esse objetivo por ao fazer a leitura da central de exceções, e encontrar o escopo da falha para exibir ao usuário.

As limitações que a ferramenta possui são a exigência de exceções encadeadas e uma classe no projeto de microserviços que faça um tratamento e salve as exceções no MongoDB. Não foi utilizado o ambiente em nuvem, apenas um computador com todos os serviços executados localmente, mas foi suficiente para fazer testes de validação e com base nos resultados da *interface gráfica*, pode afirmar que a ferramenta é útil para um processo de reprodução e correção da falha.

Para trabalhos futuros, é possível o desenvolvimento mais robusto de cada uma das etapas do projeto, focando nos pontos de melhoria destacados pela avaliação. Um ponto importante é a possibilidade de configuração para classificar as exceções na *interface gráfica*, por criticidade. Outra melhoria é a criar uma tela com grafos dos serviços por onde a requisição transitou. O foco em ambiente de produção também pode vir ser abordado, buscando melhorias no uso de máquinas virtuais e contêineres e melhores práticas para registrar as exceções.

A avaliação pode ser melhorada, seja por se criar cenários com mais microserviços e com regras de negócio baseadas em algo real e assim medir a utilidade da

ferramenta e também efetuar testes em projetos reais para coleta de informações acerca de melhorias e desempenho.

Como contribuição, o método proposto fica publicado em uma plataforma de código aberto para ser estendido ou estudado. Ademais, o protótipo apresentado permanece como uma nova referência para os desenvolvedores da área.

Referências

CARNELL, J.; SANCHEZ, I. H. *Spring Microservices in action*. Second edition. Shelter Island, NY: Manning Publications Co, 2021. ISBN 9781617296956.

CHANG, B.-M.; JO, J.-W.; HER, S. H. Visualization of exception propagation for java using static analysis. In: IEEE. *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. [S.l.], 2002. p. 173–182.

CHEN, R.; LI, S.; LI, Z. From monolith to microservices: A dataflow-driven approach. In: IEEE. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.], 2017. p. 466–475.

DEITEL, H. M.; DEITEL, H. Java. como programar. 10^a edição. Rio de Janeiro, 2010.

FU, C.; RYDER, B. G. Exception-chain analysis: Revealing exception handling architecture in java server applications. In: IEEE. *29th International Conference on Software Engineering (ICSE'07)*. [S.l.], 2007. p. 230–239.

FU, Q. et al. Where do developers log? an empirical study on logging practices in industry. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. [S.l.: s.n.], 2014. p. 24–33.

JIANG, Y.; ZHANG, N.; REN, Z. Research on intelligent monitoring scheme for microservice application systems. In: IEEE. *2020 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)*. [S.l.], 2020. p. 791–794.

KIM, M.; SUMBALY, R.; SHAH, S. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review*, ACM New York, NY, USA, v. 41, n. 1, p. 93–104, 2013.

KUROSE, J. F.; ROSS, K. W. *Redes de computadores e a internet: uma abordagem top-down*. São Paulo: Pearson, 2013. OCLC: 940078431. ISBN 9788581436777.

LAURET, A. *The design of web APIs*. [S.l.]: Simon and Schuster, 2019.

LAURETIS, L. D. From monolithic architecture to microservices architecture. In: IEEE. *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. [S.l.], 2019. p. 93–96.

- LENARDUZZI, V. et al. A dynamical quality model to continuously monitor software maintenance. In: ACADEMIC CONFERENCES INTERNATIONAL LIMITED. *The European Conference on Information Systems Management*. [S.l.], 2017. p. 168–178.
- NEWMAN, S. *Building microservices: designing fine-grained systems*. Second edition. Beijing Boston Farnham Sebastopol Tokyo: [s.n.], 2015. ISBN 9781492034025.
- NGUYEN, H. D.; SVEEN, M. *Rules for developing robust programs with java exceptions*. [S.l.], 2003.
- PINA, F. et al. Nonintrusive monitoring of microservice-based systems. In: IEEE. *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. [S.l.], 2018. p. 1–8.
- POLLARD, B. *HTTP/2 in Action*. [S.l.]: Simon and Schuster, 2019.
- PRIYA, A. *Microservices Communication: API Gateway, Service discovery server Quick Start— Part- 2*. 2020. Disponível em: <encurtador.com.br/djJX9>.
- RICHARDSON, C. *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019. OCLC: on1002834182. ISBN 9781617294549.
- SHORE, J. Fail fast [software debugging]. *IEEE Software*, IEEE, v. 21, n. 5, p. 21–25, 2004.
- URITO. 2022. Accessed: 2022-07-11. Disponível em: <<https://uri.to/>>.
- WEBBER, J.; PARASTATIDIS, S.; ROBINSON, I. *REST in practice: hypermedia and systems architecture*. 1. ed. ed. Beijing Köln: O'Reilly, 2010. (Theory in practice). ISBN 9780596805821.
- WIRFS-BROCK, R. J. Toward exception-handling best practices and patterns. *IEEE software*, IEEE, v. 23, n. 5, p. 11–13, 2006.
- ZHOU, X. et al. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, IEEE, v. 47, n. 2, p. 243–260, 2018.