Aritana Noara Costa Santos

Rastreamento automático da causa raiz da pilha de exceções em microsserviços 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: Eduardo Cunha Campos

Belo Horizonte

2022

Espaço destinado à folha de aprovação

Dedico este trabalho a Deus, minha família, professores e amigos.

Agradecimentos

Agradeço a todos que me apoiaram, que me fizeram notar meus muitos erros e perceberam alguns dos meus acertos. Percebo que 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 loga 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 neste 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 deste 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. Este trabalho é limitado à fase de projeto e 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 local environment.

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

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

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

Sumário

1 Introdução

De acordo com (RICHARDSON, 2019, p. 376-377) 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 loga uma exceção e quando isso acontece é importante a identificação da causa raiz, pois uma exceção pode ser um sintoma de uma falha do programa.

(CHEN; LI; LI, 2017, p. 466-475) menciona 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 a cidade de Venice, em uma conferência mencionada no texto original como: *workshop of software architects*, foi mencionado um nome para uma nova forma de criar aplicações: Arquitetura de Microsserviços.

Um dos fatores para o aumento da popularidade desta 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 explica que com esta arquitetura é possível desenvolver uma única aplicação como um conjunto de pequenos serviços, cada qual trabalha em seu próprio processo e conseguem ser implantados automaticamente, ao contrário de aplicações monolíticas onde uma alteração, mesmo que em uma pequena parte do sistema, implica em uma reconstrução e outra implantação completa. (CARNELL; SANCHEZ, 2021, p.15) menciona que cada serviço individual pode ser empacotado em uma máquina virtual em um provedor na nuvem e a partir desta, diversas outras instâncias podem ser

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

criadas, utilizando o conceito de escalabilidade horizontal, outra característica da Arquitetura de Microsserviços.

CARNELL cita algumas principais desvantagens, em relação à Arquitetura Monolítica, a dificuldade de depuração e de rastreamento do estado dos serviços, por ser distribuída, a arquitetura de microsserviços.

Em uma pesquisa relevante, onde (ZHOU et al., 2018) mostra 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, 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 (CARNELL; SANCHEZ, 2021, p.384). 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 (SHORE, 2004).

Para (RICHARDSON, 2019, p. 376) um serviço raramente loga uma exceção e quando isso ocorre é importante que se identifique a causa raiz, que pode indicar uma falha no programa. O trabalho de RICHARDSON também menciona que 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.

Desta forma, nesta monografia apresentamos uma abordagem para rastrear exceções em microsserviços, com foco na exceção inicial, de forma ágil, automatizada. Através da injeção de um identificador único para uma requisição que percorre

múltiplos microsserviços, e da coleta do escopo da falha, e com estas informações salvas em um banco *NoSQL* para posteriormente uma aplicação executar consultas neste 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 como a pilha ELK (Logstash, ElasticSearch e Kibana) sendo possível criar filtros e gerar visualizações (NEWMAN, 2015, p.272), é utilizada por profissionais com maior nível experiência. Há abordagens que analisam o código estaticamente para construir gráficos de propagação de exceções para auxiliar os desenvolvedores (FU; RYDER, 2007) outras propostas utilizam métricas coletadas, como latência, para gerar grafos de impacto (PINA et al., 2018a). Considerando ferramentas de monitoramento de exceções, conforme (LENARDUZZI et al., 2017) existem no mercado algumas, como a aplicação Sentry, HoneyBadger e Exceptionless. São ferramentas relativamente recentes, com 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) afirma que a maioria das companhias prefere implementar ferramentas de rastreamento e visualização próprias por questões de implementação das técnicas de arquitetura de microsserviços 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 presente estudo de caso, visa o rastreamento automático da causa raiz da exceção de *software*, em microsserviços Java, sem necessidade de fazer logs, como as ferramentas atuais, mas se valendo apenas o tratamento de exceção específico da linguagem, porém ha necessidade de criação de algumas classes no projeto para permitir salvar as informações em banco de dados, sendo uma abordagem mais intrusiva. O principal objetivo é permitir que desenvolvedor possua uma ferramenta simples que o ajude a encontrar o escopo e a causa raiz da exceção e sua pilha com maior agilidade, por não ser necessário efetuar análises de arquivos, na fase de

análise de escopo de falha. As exceções salvas em banco de dados são pesquisadas por um serviço em Node.js que acessa este banco de dados e serve como *back-end* de uma aplicação implementada em Vue.js que trabalha como *interface* gráfica para este estudo. A princípio, este tema se limita a etapa de desenvolvimento, e pode ser tema futuro uma abordagem na etapa de produção com máquinas virtuais distribuídas na arquitetura de nuvem e testes em projetos reais.

O Restante deste trabalho está organizado da seguinte maneira. O capítulo de referencial teórico aborda pesquisas e temas que outros autores realizaram que está presente neste trabalho. O capítulo de metodologia visa esclarecer de que forma este estudo foi planejado de sua execução para então ser demonstrado no capítulo que se segue, o desenvolvimento. No capítulo de desenvolvimento é possível ter uma visão acerca de todo o projeto, arquitetura, motivações e implementações. O capítulo de avaliação permite avançar nos aspectos de validação da hipótese deste estudo, explicar como que foi possível demonstrar que se chegou no resultado. Logo após, o capítulo de ameaças à Validade possibilita mencionar os pontos que invalidam a hipótese deste trabalho, ou seja, os limites nos quais o trabalho é significativo. Por último seguem-se os capítulos de conclusão e trabalho futuros e referências bibliográficas.

2 Referencial Teórico

Neste capítulo será apresentado o referencial bibliográfico 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 lista [1] 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.

2.1.1 Arquitetura *Web*

(WEBBER; PARASTATIDIS; ROBINSON, 2010, p.2) menciona que Tim Berners-Lee desenvolveu as fundações da *Web* (*Word Wide Web*) na década de 90 com a intenção de viabilizar um sistema para compartilhar documentos e tivesse algumas características como: ser distribuído, ser fracamente acoplado e ser fácil de ser utilizado. (POLLARD, 2019, p.4) 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 DOCS, 2022) [2] 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 que recebem o nome de protocolos. O protocolo mais comumente utilizado

[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.

na *Web* é o protocolo de Transferência de *Hypertexto* (*Hypertext Transfer Protocol*, ou simplesmente HTTP).

2.1.1.1 Recursos e identificadores na Web

Para (WEBBER; PARASTATIDIS; ROBINSON, 2010, p.4-6) 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 estes propósitos existe o *Uniform Resource Identifier*, URI, que seria um identificador de recursos uniforme que permite endereçar univoca 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: URI.TO/.

2.1.2 Estilo de arquitetura REST

Como abordado por ALLET (2013) [4] *Representational State Transfer* (REST), é um estilo arquitetural que descreve princípios para que recursos *Web* sejam definidos e endereçados. Características:

[3] Disponível em: <https://uri.to/faq.php>. Acesso em: 9 fev. 2022.
[4] Disponível em:<https://www.devmedia.com.br/introducao-ao-rest-ws/26964>. Acesso em: 9 fev. 2022.

- **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 destes 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.

2.1.3 Arquitetura em Nuvem

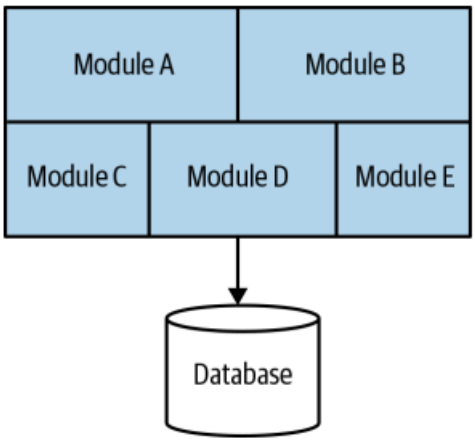
GEEKSFORGEEKS (2022) [5] define computação em nuvem como o termo designado para mencionar que o acesso e armazenamento de dados e programas acontece em servidores remotos. A arquitetura em nuvem se diz respeito aos componentes requeridos para a computação em nuvem e seus arranjos.

2.1.4 Arquitetura Monolítica

De acordo com (NEWMAN, 2015, p.12-15) sistemas que possuem uma arquitetura monolítica, no fluxo de desenvolvimento, devem ser implantados de forma única, em outra palavras, pequenas porções do sistema não podem sem implantadas independentemente. Possuem algumas vantagens em relação à arquitetura de Microsserviços, 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.

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

Figura 2 – Monolito modular



Fonte:(NEWMAN, 2015, p.13).

2.1.5 Arquitetura de Microsserviços

De acordo com (NEWMAN, 2015, p.1-11) aplicações de arquitetura de microsserviços 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 autônomas e escalabilidade de sistemas. Algumas das características dos serviços das aplicações de arquitetura de microsserviços:

- **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.

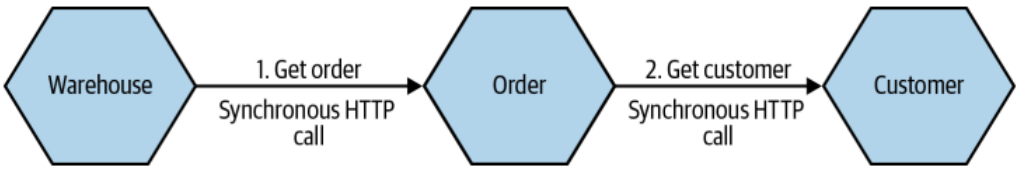
- **Escalabilidade:** cada serviço pode ser escalado independentemente quando houver necessidade.
- **Facilidade de implantação:** cada serviço pode ser criado, refatorado e então implantado independentemente dos demais.

2.1.5.1 Comunicação

Conforme (RICHARDSON, 2019, p. 42) 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; SANCHEZ, 2021, p.59) menciona 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 quem se encarrega de redirecionar a requisição para o serviço correto. (RICHARDSON, 2019, p. 121) 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

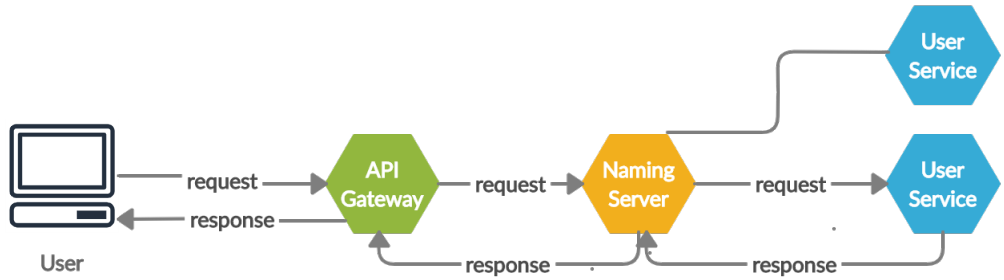


Fonte:(NEWMAN, 2015, p.22).

2.1.5.2 Computação em Nuvem

(CARNELL; SANCHEZ, 2021, p.16) menciona que microsserviços podem ser implantados em imagens de máquinas virtuais ou em contêineres virtuais (que executam dentro de uma máquina virtual) e dispostos em servidores remotos. Uma grande vantagem da computação em nuvem é o conceito de elasticidade.

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



Fonte:(PRIYA, 2020).

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, conforme mencionados na subseção anterior 2.1.5.1.

2.1.5.3 Rastreamento Distribuído

Conforme (CARNELL; SANCHEZ, 2021, p.259-261) microsserviços 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 o fluxo da transação entre serviços.

2.2 HTTP

O Protocolo de Transferência de Hipertexto (*HyperText Transfer Protocol* - HTTP) é o protocolo da camada de aplicação da *Web* que estabelece as regras de comunicação entre dois programas, cliente e servidor. HTTP utiliza o protocolo de controle de transmissão, (*Transmission Control Protocol* - TCP), como protocolo de transporte conforme (KUROSE; ROSS, 2013, p.72) mencionam.

2.3 API

API (*Application Programming Interface*) é uma *interface* de aplicação para programação que se constitui de uma coleção de protocolos de comunicação e sub rotinas que permitem programas comunicarem entre si (GEEKSFORGEEKS, 2022). [6] (LAURET, 2019, p.5) define API como *interface* exposta por algum *software*, abstraindo a complexidade da implementação.

2.3.1 API Rest

Uma aplicação Rest API, de acordo com (LAURET, 2019, p.49), 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.

2.4 Spring Boot

Segundo (CARNELL; SANCHEZ, 2021, p.5-6) *Spring* [7] é um *framework* de construção de aplicações JAVA, que possui o conceito de injeção de dependências

[6] Disponível em: <https://www.geeksforgeeks.org/introduction-to-apis> Acesso em: 5 fev. 2022.
[7] Disponível em: <https://spring.io/> Acesso em: 6 fev. 2022.

que permite externalizar relacionamentos entre objetos através de convenções e anotações de código, evitando assim que os objetos necessitem ter conhecimento de outros implicitamente, por código. *Spring Boot* [8] 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. Segundo (CARNELL; SANCHEZ, 2021, p.28) *Spring Boot* viabiliza criação de aplicações REST, seja simplificando mapeamento de estilos de verbos (HTTP) para URLs e serialização de protocolo JSON [9] e principalmente, para o foco deste trabalho, mapear exceções (Java) para códigos de erros no padrão HTTP.

2.5 Postman

Postman é uma plataforma de construção e utilização de APIs. [10]

2.6 MariaDB

Neste trabalho utilizaremos MariaDB, *OpenSource*, simplesmente para salvar dados das regras de negócio do projeto. [11]

2.7 MongoDB

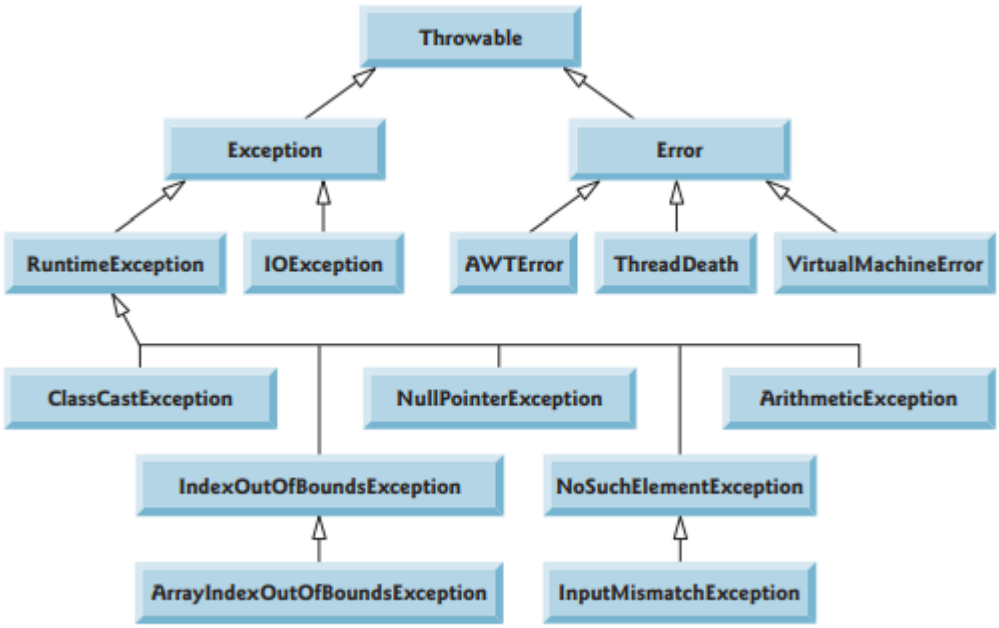
MongoDB [12] é um banco de dados não relacional, gratuito, de documentos com escalabilidade e flexibilidade. Permite armazenar dados em documentos do tipo JSON, permitindo flexibilidade para alterar os campos de documento para documento, bem como a estrutura. A opção MongoDB Atlas utilizada neste trabalho permite conectar a aplicação à sua API na nuvem, gratuitamente, mas com limitação de 512 MB de armazenamento.

[8] Disponível em: <https://start.spring.io/> Acesso em: 6 fev. 2022.
[9] Disponível em: <https://www.json.org/json-en.html> Acesso em: 6 fev. 2022.
[10] Disponível em: <https://www.postman.com> Acesso em: 6 fev. 2022.
[11] Disponível em: <https://mariadb.org> Acesso em: 6 fev. 2022.
[12] Disponível em: <https://www.mongodb.com/cloud/atlas> Acesso em: 6 fev. 2022.

2.8 Tratamento de Exceções em JAVA

(DEITEL et al., 2016, p.348-356) mencionam 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 síncronos. 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*.

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

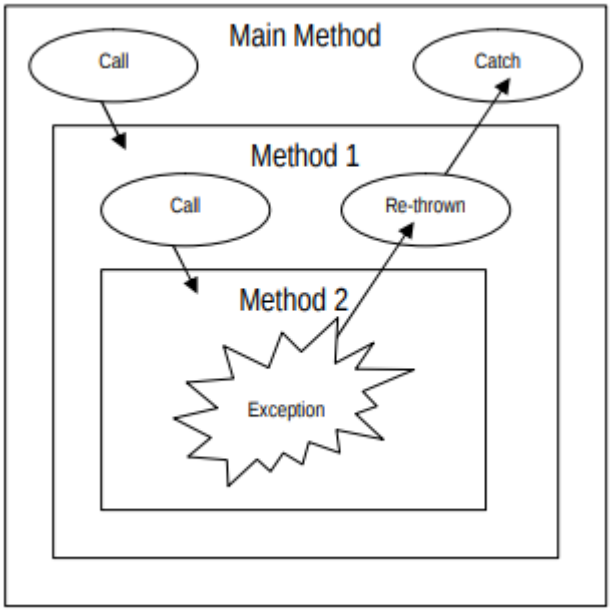


Fonte: (DEITEL et al., 2016, p.355)

O Java distingue entre 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*, (DEITEL et al., 2016; NGUYEN; SVEEN, 2003) comentam que a cláusula *throws* é utilizada quando a exceção não pode ser capturada no método onde 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, em (DEITEL et al., 2016, p.363).

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

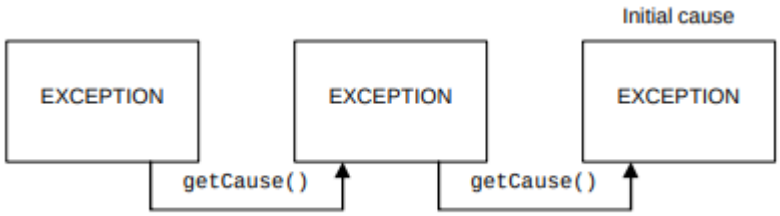


Fonte: (NGUYEN; SVEEN, 2003).

2.8.1 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*, desenvolvido para auxiliar na depuração de falhas (NGUYEN; SVEEN, 2003) conforme ilustrado na figura 7. Vale ressaltar que neste trabalho utilizaremos o método *getStackTrace()*, próprio das classes de exceção, disponibiliza o nome da classe, método e linha onde a exceção foi capturada. E como as exceções são encadeadas, neste estudo, será verificado a possibilidade de se obter os dados de todo o escopo onde uma exceção percorreu.

Figura 7 – Uma cadeia de exceções encadeadas



Fonte: (NGUYEN; SVEEN, 2003).

3 Trabalhos Relacionados

Tendo em vista uma abordagem não intrusiva para diagnosticar falhas em serviços, (PINA et al., 2018b) utilizou 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 microsserviços 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 EKL(Elasticseach + Logstash + Kibana), o *software* Zipkin e análise de arquivos de log. Concluíram que há necessidade de ferramentas de registros e visualização mais inteligentes. Pois, depende muito da experiência do desenvolvedor inferir se o atual estado de microsserviços pode revelar uma falha.

Ao analisar e revisar as diretrizes básicas e padrões de tratamento de exceções (WIRFS-BROCK, 2006) demonstra em seu estudo que o uso adequado do tratamento de exceções de falhas é crucial para a compreensão, evolução e refatoração em *software*.

(FU; RYDER, 2007) apresentam 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 destes. 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. Este estudo gera diversos gráficos de dependência que realça o fluxo de exceções entre componentes.

(CHANG; JO; 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 checadas, 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; 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 significante melhoria no apontamento de causas raízes de falha.

(LENARDUZZI et al., 2017) cita 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 destas ferramentas classificam o problema imediatamente e que não provêm percepções para auxiliar desenvolvedores encontrar causas eficientemente. Para se utilizar tais ferramentas é necessário incluir logs específicos no código para se fazer o registro das exceções.

(JIANG; ZHANG; REN, 2020)JIANG (2020) utilizou uma pilha EKL (Elasticseach + Logstash + Kibana) para coletar dados de logs, agregar em 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.

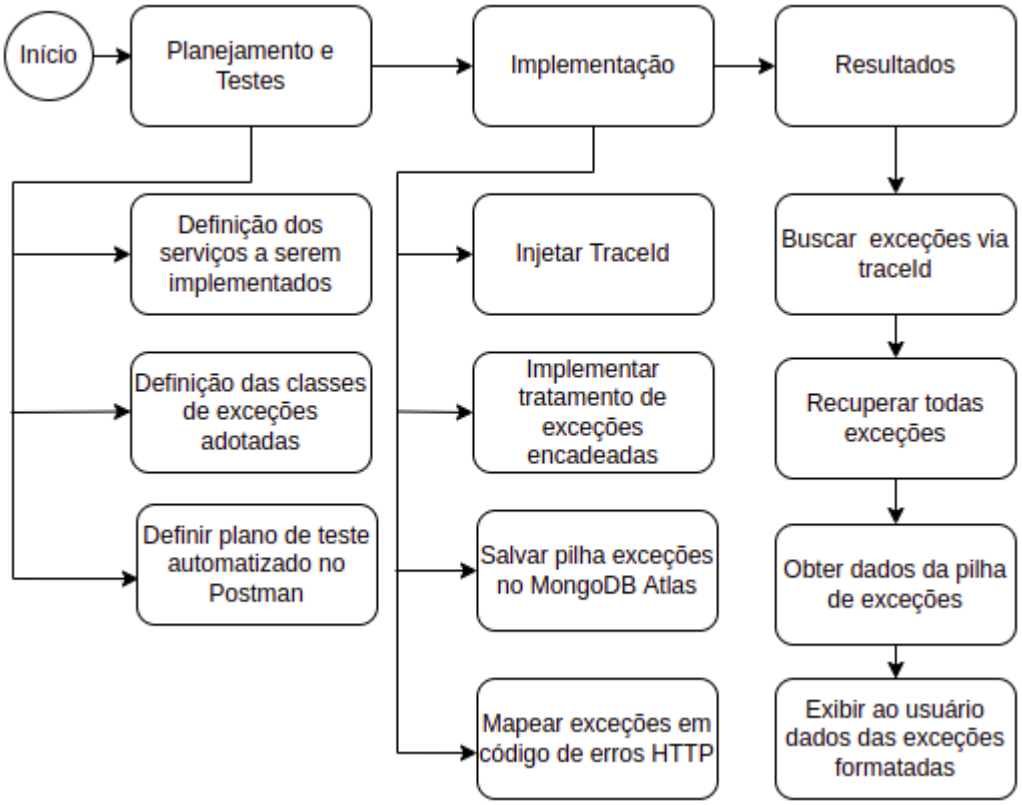
Com o foco na análise de falhas de sistemas e a busca pela causa raiz, é possível dizer que os trabalhos relacionados relatados neste capítulo buscam analisar o cenário da arquitetura e as maneiras de se obter informações sobre falhas, e como encontrar a sua causa raiz, seja coletando métricas através de registro de logs e realizar análise destes registros através de ferramentas. No campo de tratamento de exceções, algumas pesquisam realizaram a análise estática do código para se 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 deste 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, e o código no GitHub, podendo ser mais uma contribuição para esta área de estudo ao lado dos trabalhos encontrados na literatura.

4 Metodologia

A metodologia empregada no presente trabalho tem o intuito de criar uma solução para verificar se a hipótese deste trabalho é viável e as etapas podem ser visualizadas na figura 8. O projeto implementado na linguagem Java encontra-se no GitHub [1].

Figura 8 – Objetivos gerais da metodologia



Fonte: Elaborada pelo autor.

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

4.1 Objetivos

Os principais objetivos são:

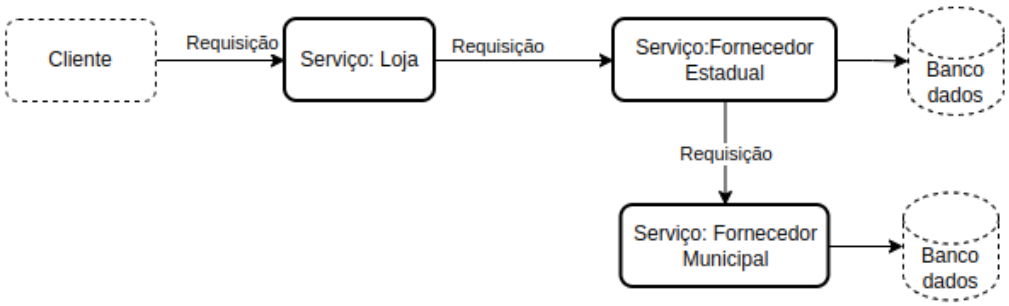
- **Identificador Único**: injetar um identificador em cada transação. Para tanto é necessário utilizar a biblioteca Spring Cloud Sleuth. [2]
- **Pilha de exceções**: tratar e salvar a pilha de exceções encadeadas que possam ocorrer em algum serviço em um local único, o banco de dados MongoDB Atlas. [3]
- **Serviço central**: banco de dados MongoDB Atlas, um serviço em nuvem, que será útil para armazenar os registros de pilhas de exceções. [4]
- **Interface gráfica**: aplicação para acessar o serviço central e 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 números das linhas. Disponibilizar ao desenvolvedor um *link* para classe onde foi lançada e exceção raiz na plataforma GitHub. Foi desenvolvida em VUE.js. [5]

4.2 Planejamento

- **Definição dos serviços a serem implementados**: criação de três serviços, de acordo com a figura 9.
- **Definição do número de porta**: cada serviço precisa ter seu número de porta publicado no arquivo de configuração do projeto e ter um código é útil para a fase de testes, de acordo com a tabela 1.

[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.

Figura 9 – Serviços Loja, Fornecedor Estadual e Fornecedor Municipal



Fonte: Elaborada pelo autor.

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

Tabela 1 – Informações que identificam um serviço.

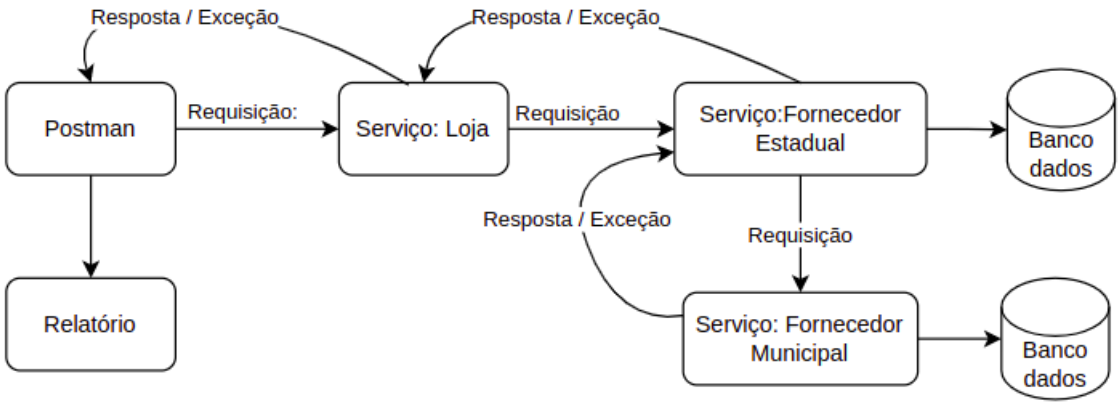
- **Definição das classes de exceções adotadas**: foi definido um grupo de classes de exceções, para serem utilizadas, individualmente, como causa raiz na fase de testes. Cada exceção será mapeada a um código erro HTTP, de acordo com a tabela 2.

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

Tabela 2 – Grupo de exceções utilizadas para serem causa raiz na fase de testes.

- **Definir plano de teste**: será utilizado o Postman como simulador de requisições de cliente de acordo com a figura 10.

Figura 10 – Cenário de teste por simulação de requisição de cliente via Postman

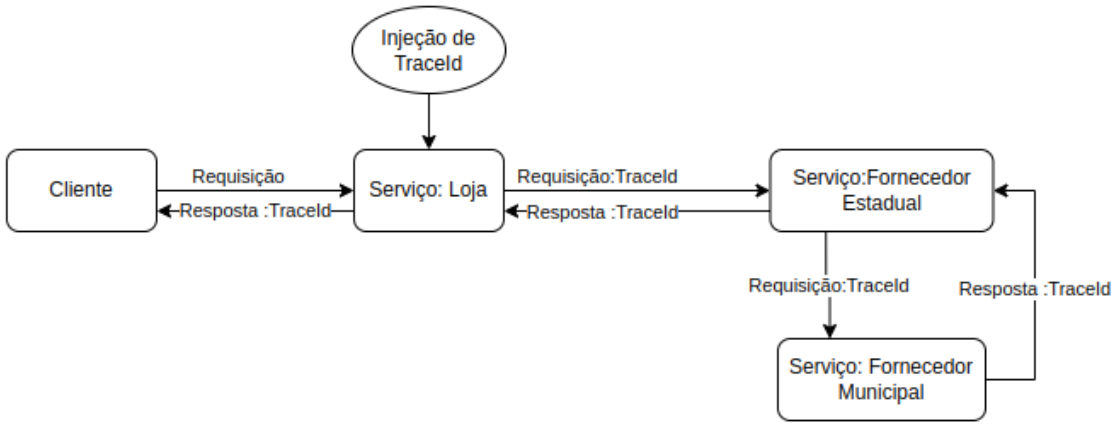


Fonte: Elaborada pelo autor.

4.3 Implementação

- **Injetar Identificador Único da transação - TraceID**: injeção de traceId, de acordo com a figura 11.

Figura 11 – Injeção de traceId na requisição.



Fonte: Elaborada pelo autor.

- **Implementar tratamento de exceções encadeadas e Mapear exceções em código de erros (HTTP)** : em cada declaração *try...catch*, 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:

Figura 12 – 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;
        }
    }
```

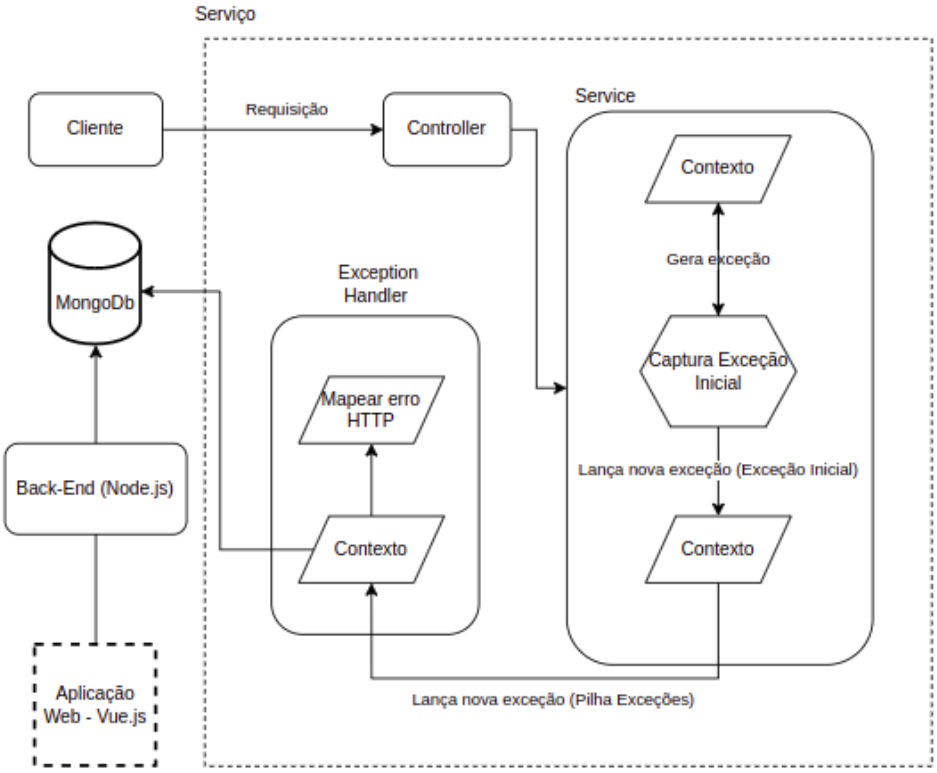
Fonte: Elaborada pelo autor.

- **Salvar pilha exceções no MongoDB Atlas**: A pilha de exceção é tratada para ser disponibilizada em formato JSON e então é salva no MongoDB Atlas, conforme a figura 13.

A figura 14 demonstra 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.

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

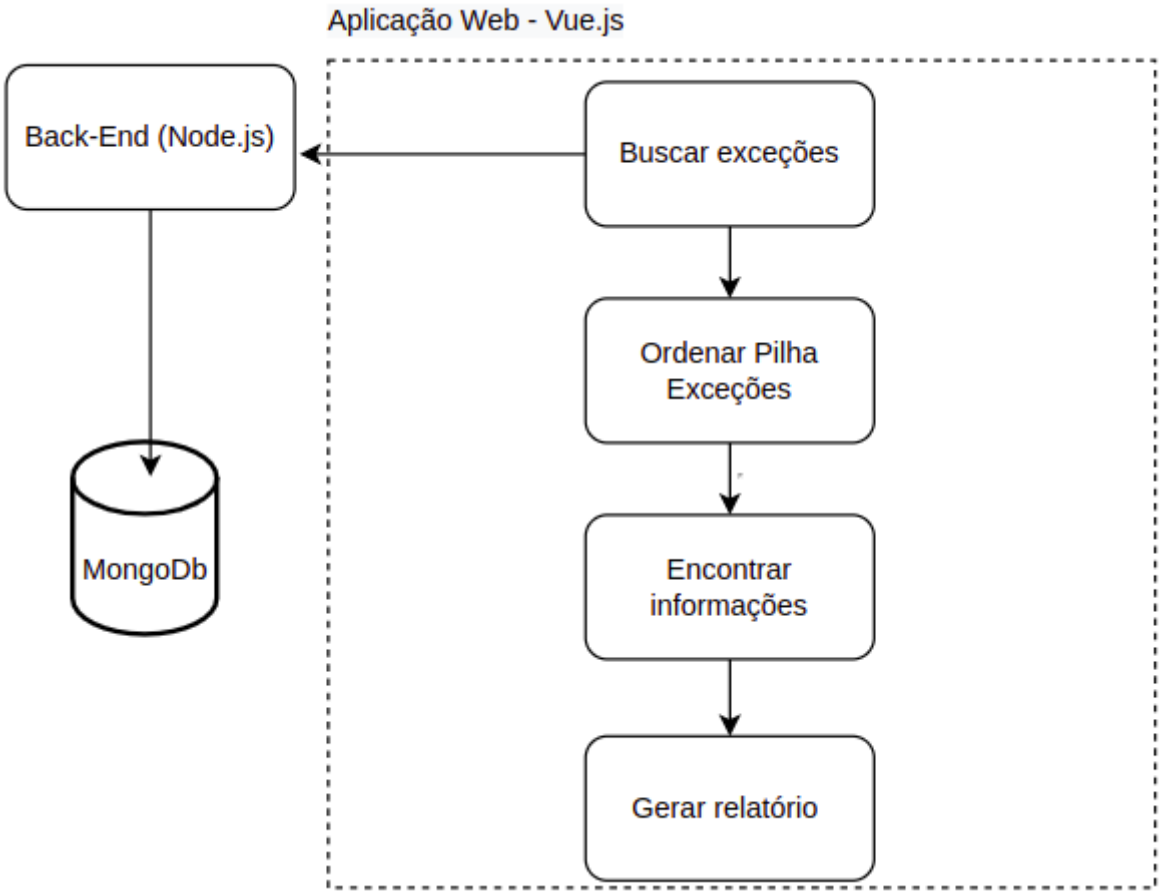
Figura 13 – Pilha de exceção é tratada e enviada ao MongoDB Atlas.



Fonte: Elaborada pelo autor.

O modelo para parte gráfica que servirá de apoio para o desenvolvedor encontrar as exceções está ilustrada figura 14. O *Back-end* irá suprir as requisições da aplicação Web. A aplicação Web irá ter a função de fornecer ao usuário o máximo de informação a respeito do escopo da falha com o mínimo de esforço do usuário.

Figura 14 – Aplicação para resolver e gerar informações de falhas no sistema.

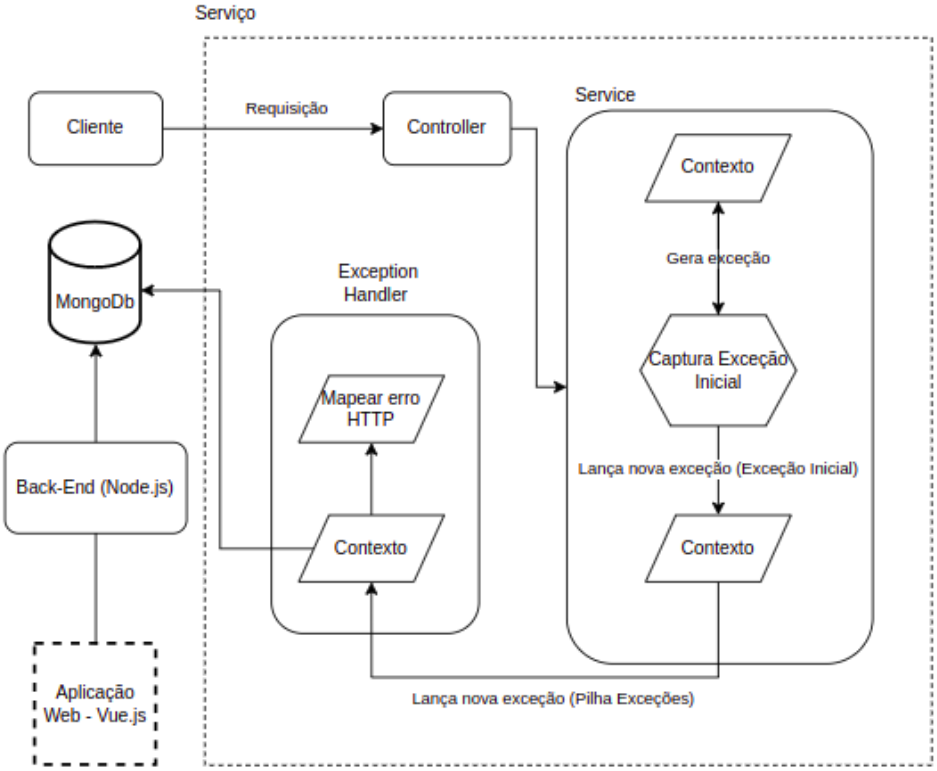


Fonte: Elaborada pelo autor.

5 Desenvolvimento

O projeto completo pode ser acessado no GitHub [1]. Conforme *link* no rodapé da página.

Figura 15 – Arquitetura.



Fonte: Elaborada pelo autor.

A arquitetura ilustrada na figura 15 foi esboçada para o propósito deste trabalho de gerenciar as exceções ocorridas nas requisições entre microsserviços, 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

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

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 estes 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 destas definições:

Componente	**Responsabilidade**
Serviço	Arquitetura dos serviços.
MongoDb Atlas	Camada de persistência para exceções.
Back-End (TrackerCentral)	Camada de interface entre o MongoDb e o front-end.
Front-End (Traker-v1)	Permite visualização gráfica e organizada das exceções.

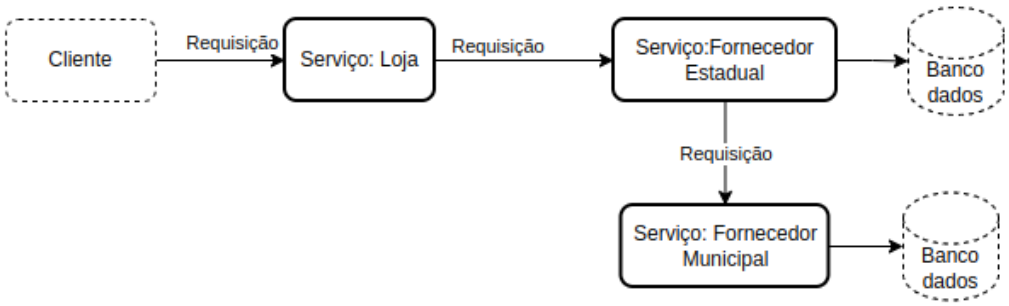
Tabela 3 – Componentes da arquitetura e responsabilidades.

5.1 Microsserviços

A seção 2.1.5 definiu arquitetura de microsserviços [2] e para a abordagem deste trabalho utilizaremos três microsserviços pequenos e responsáveis pelas tarefas definidas na tabela 4.

[2] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud> Acesso em: 6 fev. 2022.

Figura 16 – Serviços Loja, Fornecedor Estadual e Fornecedor Municipal



Fonte: Elaborada pelo autor.

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.

Estes serviços foram implementados utilizando a linguagem Java e o *framework Spring Boot* abordado na seção A seção 2.4. Além disso, foram utilizadas diversas bibliotecas para conforme seção 5.1.2 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.

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

Diversas bibliotecas foram utilizadas como dependências [3] [4] [5] [6] [7]

do *Spring Boot*:

- ***Spring Cloud Sleuth***: possibilita injetar um identificador único (traceID) na transação que percorre vários microsserviços.CARNELL (2017, p. 2593-261).
- ***Spring Cloud Open Feign***: cliente HTTP usado para fazer chamadas síncronas entre APIs expostas pelos microsserviços.
- ***Spring cloud***: providencia ferramentas para construção de padrões comuns em sistemas distribuídos.

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

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

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

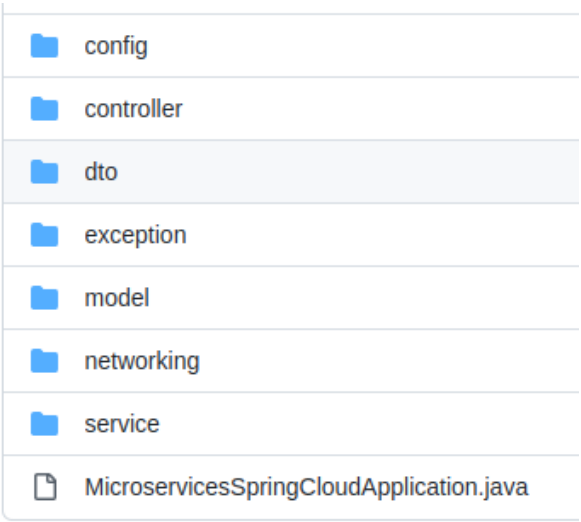
[3] Disponível em:<https://www.tutorialspoint.com/spring_cloud/spring_cloud_synchronous_communication_with_feign.htm> Acesso em: 6 fev. 2022.
[4] Disponível em: <https://www.javatpoint.com/spring-cloud> Acesso em: 6 fev. 2022.
[5] Disponível em: <https://www.baeldung.com/intro-to-project-lombok> Acesso 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> Acesso em: 6 fev. 2022.

5.1.3 Arquitetura

Os serviços [8] possuem basicamente a mesma estrutura de pacotes como indicado na figura 17. A classe principal *MicroservicesSpringCloudApplication* possui o método *public static void main()* que inicializa a aplicação *Spring*, conforme seção 2.4 , e possui a definição da *url* de autenticação com o mongoDB Atlas.

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

Figura 17 – modelagem de pacotes Java



Fonte: Elaborada pelo autor.

5.1.3.1 Config

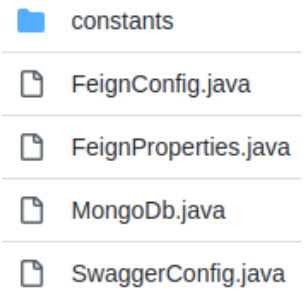
O pacote **Config** [10] possui a responsabilidade de agrupar classes cujo objetivo é de configuração das bibliotecas utilizadas no projeto e do próprio projeto.

[8] Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud> Acesso em: 6 fev. 2022.

[9] Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado> Acesso em: 6 fev. 2022.

[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> Acesso em: 6 fev. 2022.

Figura 18 – pacote config



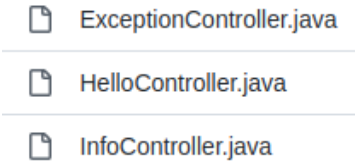
Fonte: Elaborada pelo autor.

A figura 18 ilustrada o conteúdo do pacote. As classes configuram bibliotecas que foram definidas na seção 5.1.2.

5.1.3.2 Controller

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

Figura 19 – pacote Controller



Fonte: Elaborada pelo autor.

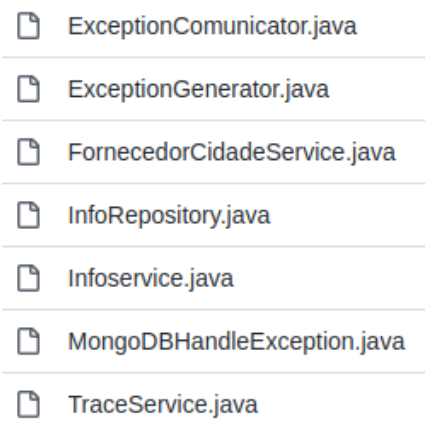
[11] Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/controller> Acesso em: 6 fev. 2022.

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

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]

Figura 20 – pacote Service



Fonte: Elaborada pelo autor.

[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> Acesso em: 6 fev. 2022.

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

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 21 – pacote Dto

Fonte: Elaborada pelo autor.

5.1.3.5 Exception

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

Figura 22 – pacote Exception

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> Acesso em: 6 fev. 2022.

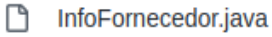
[16] Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/exception> Acesso em: 6 fev. 2022.

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

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 a anotações *@Entity e @Table*. [19]

Figura 23 – pacote Model

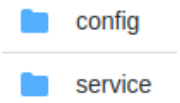


Fonte: Elaborada pelo autor.

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* possuí 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 24 – pacote NetWorking



Fonte: Elaborada pelo autor.

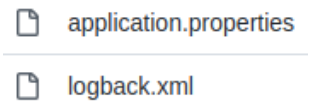
[18] Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/model> Acesso em: 6 fev. 2022.
[19] Disponível em: <https://www.baeldung.com/jpa-entity-table-names> Acesso em: 6 jun. 2022.
[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> Acesso em: 6 fev. 2022.

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 26 neste 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 desejar utilizar recursos. Além disso, as configurações com o banco de dados MariaDB.

Figura 25 – Application.properties



Fonte: Elaborada pelo autor.

[21] Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/microservices-spring-cloud/cloud-entregador-estado/src/main/resources> Acesso em: 6 fev. 2022.

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

Figura 26 – 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 7 no serviço com código respectivo à tabela 6.

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

Tabela 6 – Informações que identificam um serviço.

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

Tabela 7 – Grupo de exceções utilizadas para serem causa raiz na fase de testes.

5.1.3.10 Principais Resultados Alcançados

- **Injetar Identificador Único da transação - TraceID**: como mencionado na seção 2.1.5.3, 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 27, 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, através da classe *TraceService.java* disponível em todos os serviços. [24]

[23] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/microservices-spring-cloud/cloud-entregador-estado/pom.xml> Acesso em: 6 fev. 2022.

[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> Acesso em: 6 fev. 2022.

Figura 27 – TraceID: ID único da requisição/resposta HTTP

Fonte: Elaborada pelo autor.

- **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 *Spring Boot* é possível anotar uma classe para ser responsável por coletar todas as exceções que vier ocorrer em todo o programa, antes de ser enviada na requisição, (*@RestControllerAdvice*). Nesta camada, as exceções são mapeadas para códigos de respostas HTTP, sendo tratadas para serem exibidas em um formato definido no projeto. A figura 27 exibe o formato definido neste projeto para exibição de exceções como respostas de requisições HTTP e a figura 28 exibe um trecho do código da classe *CentralExceptionHandler*. [26]

- **Persistir Exceções no MongoDB Atlas**: classe *CentralExceptionHandler*, como foi mencionado, centraliza as exceções e então consome métodos da classe de serviço *MongoDBHandleException* que possui a lógica para salvar a pilha de exceção no MongoDB Atlas. [27]. Cada serviço consegue acessar a

[25] Disponível em: <https://www.geeksforgeeks.org/chained-exceptions-java/> Acesso em: 6 fev. 2022.
[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/> Acesso em: 6 fev. 2022.
[27] Disponível em: <https://github.com/aritana/tcc2-exceptionTracker/blob/master/

Figura 28 – 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.responseError == 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.

mesma aplicação do mongoDB Atlás, sendo um serviço baseado em nuvem, com versão gratuita limitada a 512 MB de armazenamento.

Back/microservices-spring-cloud/cloud-entregador-estado/src/main/java/alura/br/microservicesspringcloud/service/MongoDBHandleException.java> Acesso em: 6 fev. 2022.

- **Geração de exceções em um serviço**: com a ferramenta *Postman* [28] é possível simular requisições que geram exceções como causa raiz, conforme tabela 7, nos serviços desejados, como tabela 6, 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 6; e deseja neste serviço provocar uma falha (e.g divisão por zero) que gere uma exceção "NullPointerException", cujo código é 3, conforme tabela 7, deve prosseguir com a seguinte URL:

  http://localhost:{numeroPortaServico}/fornecedorCidade/exception/{codigoExcecao}/{codigoServico} conforme rota mencionada na seção 5.1.3.9.

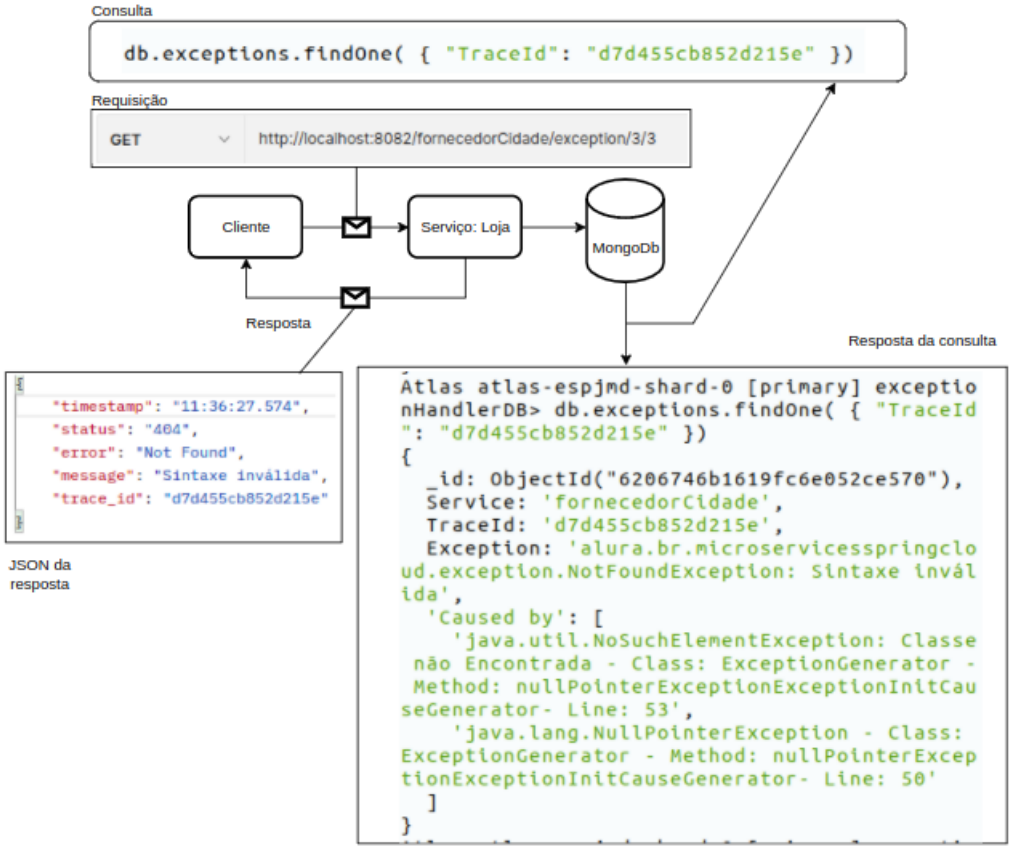
  que adaptada, conforme códigos nos campos codigoExcecao e codigoServico fica: http://localhost:8080/fornecedorCidade/exception/3/3.

  O cliente só se comunica com o serviço "Loja", cuja porta é 8080, conforme ilustra a figura 16 e tabela 6.

  Desta forma, o resultado para esta solicitação pode ser visualizado a seguir, conforme figura 29

[28] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Postman> Acesso em: 6 fev. 2022.

Figura 29 – Resultado para uma requisição de teste

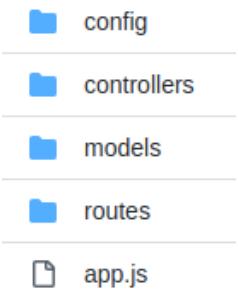


Fonte: Elaborada pelo autor.

5.2 Back-end trackerCentral

A API TrackerCentral [29] foi desenvolvida em *Node.js*, se conecta ao MongoDB Atlas e disponibiliza *endpoints* à *interface* gráfica, que no que lhe concerne irá providenciar os relatórios gráficos ao usuário final.

Figura 30 – TrackerCentral



Fonte: Elaborada pelo autor.

5.2.0.1 Conexao MongoDB Atlas

Para se conectar ao banco MongoDB Atlas, utilizou a biblioteca Mongoose [30], configurada no arquivo *dbConnect*. [31]

5.2.0.2 *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 31, obtido pela ferramenta *Postman*.

[29] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Back/trackerCentral> Acesso em: 6 fev. 2022.
[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> Acesso em: 6 fev. 2022.
[32] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Back/trackerCentral/src/models/Exceptions.js> Acesso em: 6 fev. 2022.
[33] Disponível em:<https://mongoosejs.com/docs/guide.html> Acesso em: 6 fev. 2022.

Figura 31 – *Schema* do objeto Exceção


```
{
    "_id": "6296e026151d3c3f160cae06",
    "service": "fornecedorCidade",
    "traceId": "49dd03f55d992a92",
    "exception": "alura.br.microservicesspringcloud.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/
        microservicesspringcloud/service/ExceptionGenerator.java"
},
```


Fonte: Elaborada pelo autor.

5.2.0.3 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.

5.2.0.4 Principais Resultados Alcançados

O *O back-end* irá disponibilizar através de sua API os *endpoints* para o *front-end*. O *front-end* irá utilizar apenas duas das rotas disponibilizadas pelo

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

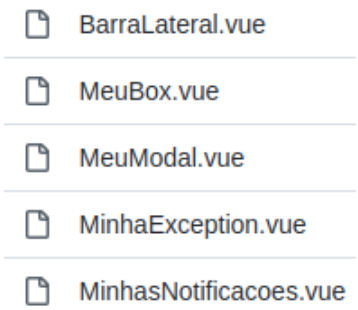
5.3 Front-end tracker-v1

A API tracker-v1 [34] 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.

5.3.0.1 Componentes

Componentes são instâncias reutilizáveis do Vue. [36] Os componentes necessários para criação do esquema visual estão ilustrados na figura 32. O componente mais importante, MinhaException.vue, representa uma exceção no topo da pilha de exceções encadeadas obtidas do *back-end*.

Figura 32 – TrackerCentral



Fonte: Elaborada pelo autor.

[34] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Front/tracker-v1> Acesso em: 6 fev. 2022.
[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.

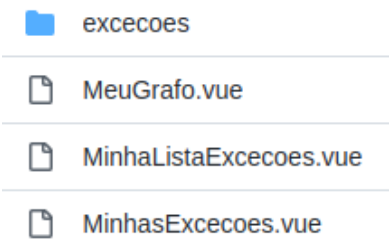
5.3.0.2 Rotas

As rotas, descritas no arquivo router/index.ts [37], define as rotas da aplicação e permite a transição entre *views* conforme interação com o usuário.

5.3.0.3 Views

As *views* [38] permite criar visualizações a partir de arranjo de componentes sendo roteáveis [39]. A figura 33 ilustra as *views* do projeto. O arquivo MeuGrafo.vue não é utilizado neste trabalho, se trata de uma experiência para trabalhos futuros, que visa utilizar grafos para mapear os serviços do fluxo da exceção.

Figura 33 – TrackerCentral

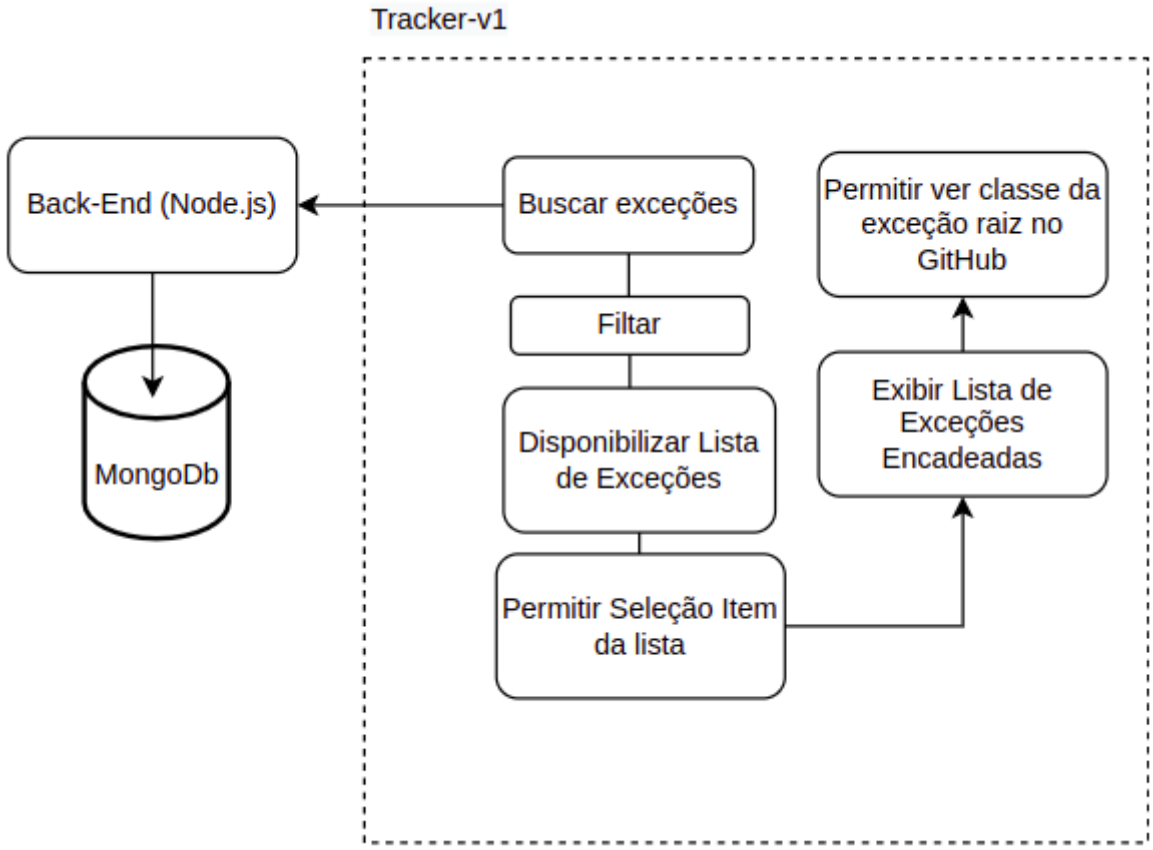


Fonte: Elaborada pelo autor.

[37] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/blob/master/Front/tracker-v1/src/router/index.ts> Acesso em: 6 fev. 2022.
[38] Disponível em:<https://github.com/aritana/tcc2-exceptionTracker/tree/master/Front/tracker-v1/src/views> Acesso em: 6 fev. 2022.
[39] Disponível em:<https://router.vuejs.org/guide/essentials/named-views.html> Acesso em: 6 fev. 2022.

5.3.0.4 Principais Resultados Alcançados

Figura 34 – Tracker-v1.



Fonte: Elaborada pelo autor.

A figura 14 descrita no capítulo de 4, descreve a ideia do projeto e sofreu alguns ajustes que serão explicados a seguir e ilustrado na figura 34.

- **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.

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

- **Filtrar**: Toda exceção encadeada tem no mínimo dois níveis, pois como definido na seção 5.1.3.10, cada serviço possui uma camada *Exception Handler* que faz o mapeamento das exceções internas para um modelo de reposta mapeado para o protocolo HTTP. Sendo essa exceção mais externa declarada no modelo de objeto *JSON* como **exception** e a pilha de exceções declarada no mesmo modelo como **causedBy**, um *array* com todas exceções encadeadas inerentes ao código do projeto, como pode ser observado na figura 31.

  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 35

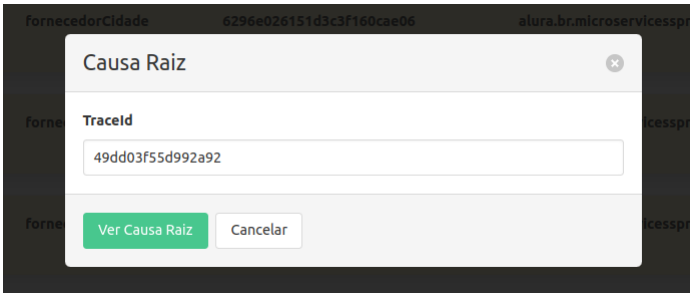
Figura 35 – Tracker-v1: lista de exeções.

Fonte: Elaborada pelo autor.

- **Permitir seleção de item da Lista de Exceções**: As exceções listadas são clicáveis, e o usuário pode selecionar alguma, conforme figura 36

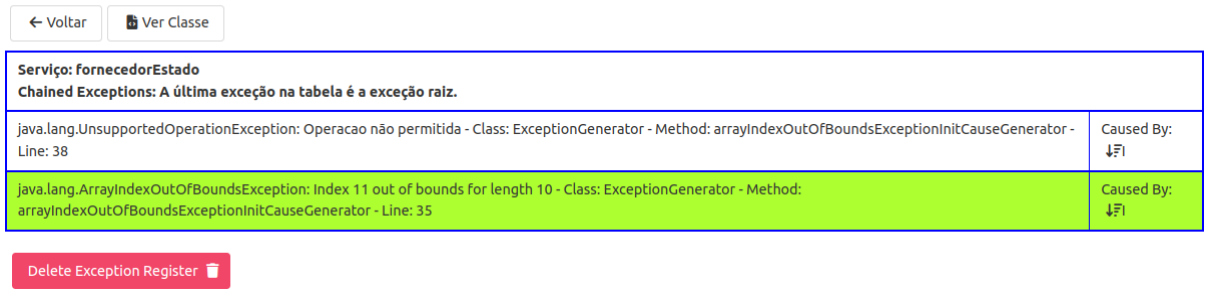
- **Exibir lista de exceções encadeadas**: A exceção selecionada possui o encadeamento de outras exceções que devem ser exibidas, com a exceção raiz em destaque no fim da tabela. Algo notável é que cada exceção possui informações do escopo da falha, o nome da classe, o nome do método, e a linha onde ocorreu a exceção, como ilustrado na figura 37.

Figura 36 – Tracker-v1: seleção de exceção para verificar sua causa raiz.



Fonte: Elaborada pelo autor.

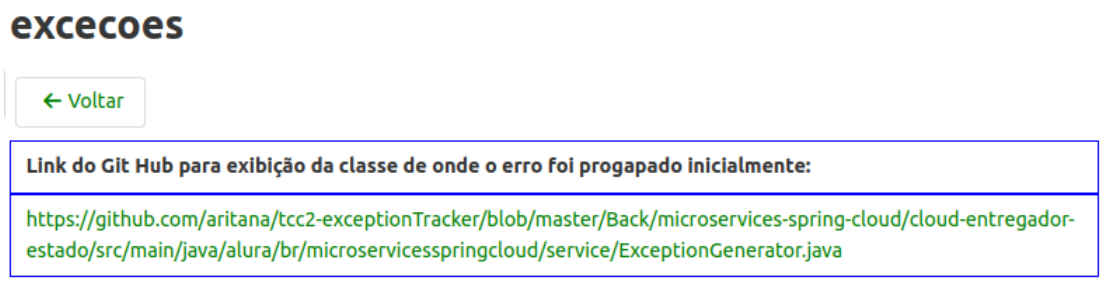
Figura 37 – Tracker-v1: exceções encadeadas com causa raiz em destaque.



Fonte: Elaborada pelo autor.

- **Permitir ver o escopo da exceção no GitHub**: É possível ao clicar no botão *Ver Classe*, ilustrado na figura 37 e ser direcionado para o código no GitHub, após clicar no *link* exibido, como exibido na figura 38. 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 38 – Tracker-v1: link para acessar o escopo no GitHub.



Fonte: Elaborada pelo autor.

5.3.0.5 Informações acerca do projeto de estudo de caso

Durante o semestre, foram realizados diversos cursos das áreas de estudo do trabalho de conclusão de curso para buscar melhor embasamento para o projeto além da experiência prática na exerção de estágio não supervisionado. Acerca dos conceitos aprendidos, seja da teoria ou do código utilizado durante o curso, foi aproveitado neste estudo e a seguir será exibida a lista dos cursos para verificação:

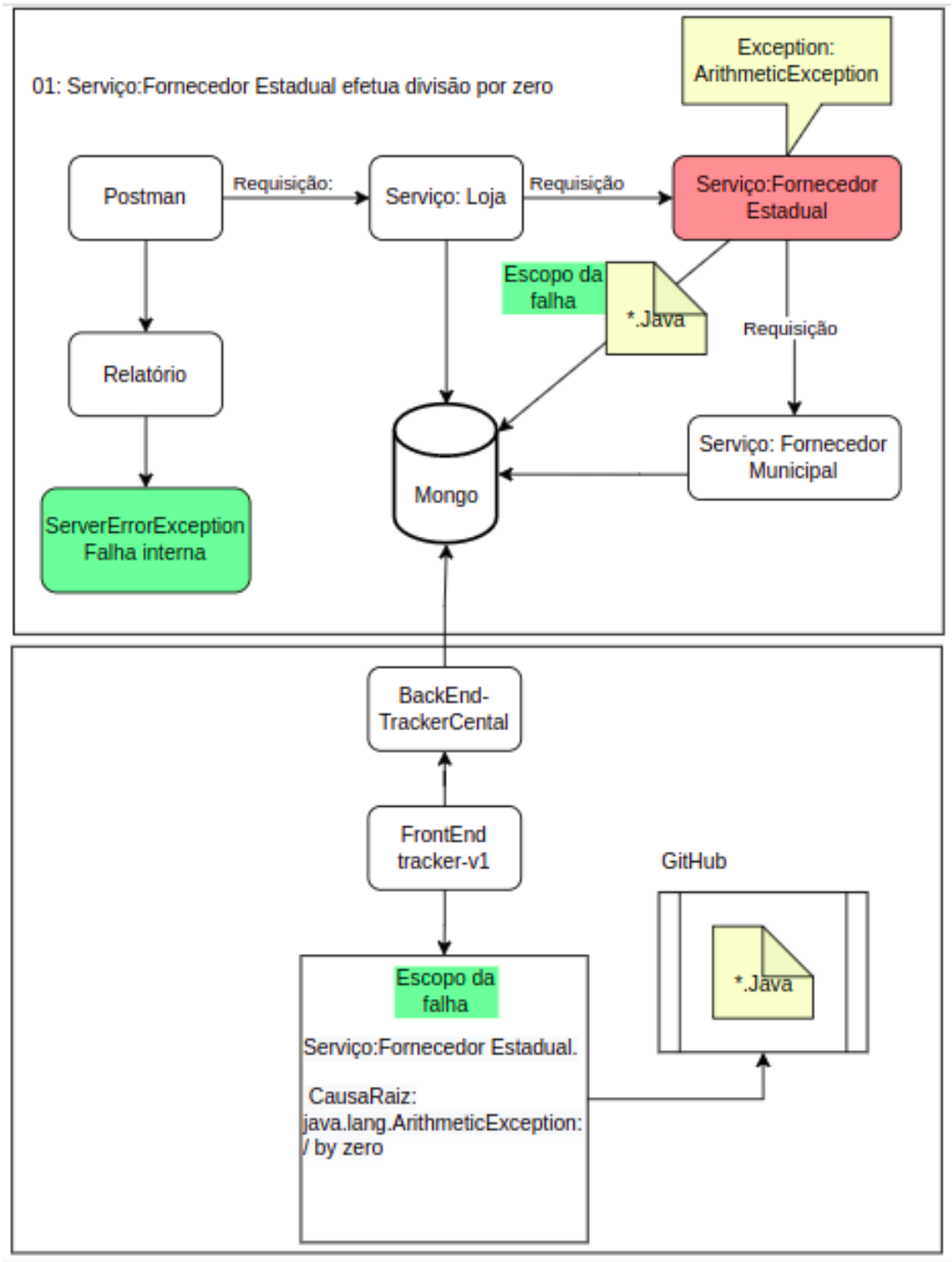
- **Microsserviços Java**:os cursos relevantes para este tema foram: *Spring Boot API REST: construa uma API* [41] que concedeu conceitos para a criação de uma API Rest e *Microservices com Spring Cloud: Registry, Config Server e Distributed Tracing* [42] que permitiu a compreensão da comunicação entre serviços e a injeção de identificador único nas requisições.
- **MongoDB e Node.js**: os cursos relevantes para este 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 este 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 do trabalho deste trabalho consiste em verificar se o escopo da falha definida em um microsserviço foi avaliada com sucesso pela *interface gráfica*. A figura 39 ilustra o processo de validação. O processo de geração de exceção, mencionado na seção 5.1.3.10, permite possibilidade lançar uma exceção em qualquer serviço escolhido, automaticamente, por uma requisição customizada no Postman. Os testes para validar se mostraram eficazes para o objetivo deste trabalho, pois as exceções provocadas são observadas na *interface* gráfica acuradamente.

Figura 39 – Tracker-v1: exceções encadeadas com causa raiz em destaque.



Fonte: Elaborada pelo autor.

7 Ameaças à validade

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 40.

Figura 40 – 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.

Uma segunda limitação é que as exceções lançadas necessitam estar declaradas na classe CentralExceptionHandler.java [1]. Pois, esta classe captura automaticamente todas exceções lançadas no serviço antes de enviar a enviar resposta da requisição, e a captura do escopo, como descrito na seção 5.2.0.4 e ilustrado na figura 41.

[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.

Figura 41 – 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.responseError == 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.responseError;
        }
        logger.debug("ServerErrorException {}", exception.getMessage());
        mongoDBHandleException.saveException(exception);
        return responseError;
    }
```

Fonte: Elaborada pelo autor.

A terceira limitação está relacionada a maneira intrusiva que a abordagem exige, se faz necessário criar classes para adequar a exceção posterior ao salvamento no banco de dados e a classe que registra essa exceção no banco de dados.

A quarta limitação é inerente à estratégia de avaliação do protótipo. Não foi possível, devido ao tempo, implementar 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 menciona a literatura.

8 Conclusões e Trabalhos Futuros

O presente trabalho cumpriu ao objetivo de centralizar as exceções, e então encontrar o escopo da falha, em um sistema distribuído de microsserviços, de forma simples, sem necessidade de logs adicionais no código. Limitado ao ambiente de desenvolvimento. Como dito anteriormente, esta abordagem é intrusiva, e o projeto necessita ser alterado para receber classes dedicadas a função de tratar e registar exceção no banco central.

Com base nos resultados da *interface gráfica*, com o fechamento com um *link* ao código do gitHub, foi significativo. A ferramenta demonstra ser útil ao apoio de desenvolvedores no 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 registras as exceções.

A avaliação pode ser melhorada, seja por se criar cenários com mais microsserviços 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. et al. Java: como programar. Biblioteca Hernán Malo González, 2016.

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*. S??o 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.

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.

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.