UNIVERSIDADE ESTADUAL PAULISTA JÚLIO DE MESQUITA FILHO
FACULDADE DE CIÊNCIAS
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Trabalho de Conclusão de Curso

Rafael Colin Rios

Distrace:
Um software sobre observabilidade e visualização

Novembro, 2024

UNIVERSIDADE ESTADUAL PAULISTA JÚLIO DE MESQUITA FILHO
FACULDADE DE CIÊNCIAS

SISTEMAS DE INFORMAÇÃO

Rafael Colin Rios

Distrace:
Um software sobre observabilidade e visualização

Trabalho apresentado como exigência parcial para a conclusão do Curso de Bacharelado em Sistemas de Informação da Faculdade de Ciências – UNESP Campus Bauru.

Orientador: Prof. Dr. Higor Amario de Souza

Bauru, Novembro, 2024

C696d

Colin Rios, Rafael
Distrace : Um Software sobre observabilidade e visualização / Rafael Colin Rios. -- Bauru, 2024
44 p.

Trabalho de conclusão de curso (Bacharelado - Sistemas de Informação) - Universidade Estadual Paulista (UNESP), Faculdade de Ciências, Bauru
Orientador: Higor Amario de Souza

1. Compreensão de programas. 2. Visualização de programas. 3. Métricas de programas. 4. Manutenção de programas. 5. Microsserviços. I. Título.

RAFAEL COLIN RIOS

DISTRACE: UM SOFTWARE SOBRE OBSERVABILIDADE E VISUALIZAÇÃO

Trabalho de Conclusão de Curso apresentado à Faculdade de Ciências do Campus Bauru, Universidade Estadual Paulista Júlio de Mesquita Filho, como parte dos requisitos para obtenção do título de Bacharelado em Sistemas de Informação.

Bauru, 25 de novembro de 2024

BANCA EXAMINADORA

_____
Prof. Higor Amario de Souza
Orientador
Faculdade de Ciências - Unesp/Bauru
Departamento de Computação

_____
Prof. José Remo Ferreira Brega
Faculdade de Ciências - Unesp/Bauru
Departamento de Computação

_____
Prof. Márcia A. Zanoli Meira e Silva
Faculdade de Ciências - Unesp/Bauru
Departamento de Computação

AGRADECIMENTOS

Gostaria de agradecer imensamente aos meus pais, Ana e Sandro, e à minha irmã, Laura, que estiveram comigo em cada passo desta jornada, oferecendo apoio constante e formando as bases para eu me tornar a pessoa que sou hoje.

À minha namorada, Melissa, cuja dedicação e empenho são uma inspiração para mim. Sou muito feliz ao seu lado e grato pela honra de estar com você neste momento tão importante da minha vida.

Aos meus amigos, Giovana, Annanda e Vitor, que compartilharam as experiências acadêmicas comigo e me ajudaram a enfrentar os desafios da vida universitária.

Gostaria de expressar meus sinceros agradecimentos ao meu professor orientador, Higor Amario de Souza, por todo o suporte e orientação fornecidos ao longo do desenvolvimento deste projeto, e também a todo o corpo docente, com quem tive o prazer de aprender ao longo deste percurso.

RESUMO

Ao analisar o atual cenário de criação de software é possível notar um grande aumento na complexidade no processo de desenvolvimento, tanto para construção de código quanto para gestão da engenharia e arquitetura por trás do produto final. Sendo possível, graças a diferentes tipos de arquiteturas para se construir uma aplicação, agregar de forma colaborativa diferentes linguagens de programação em um mesmo projeto, fornecendo diferentes tipos de serviço. Dessa maneira, a criação de software se tornou algo que vai além dos programadores envolvidos no projeto, sendo necessário a participação de indivíduos que cuidam do design, da venda, da manutenção do sistema, seja através da correção de problemas ou então da própria infraestrutura do sistema em si. Pensando nisso, surge a necessidade de abstrair toda essa visão de como o software está estruturado, para gerar uma visualização mais acessível a todos os que estão envolvidos no processo de se desenvolver um software. Por esta razão, visando contribuir para a evolução do processo de se produzir um software e na colaboração do maior número possível de pessoas nesta tarefa, este software tem como função a criação de uma aplicação que obtém, trata e metrifica os dados a partir de uma aplicação externa, gerando uma visualização para representar as relações e estruturas dentro de um projeto de software.

Palavras-chave: *Compreensão de programas; visualização de programas; métricas de programas; manutenção de programas; microsserviços.*

ABSTRACT

In analyzing the current software development landscape a significant increase in complexity can be observed in the development process, both in code construction and in managing the engineering and architecture behind the final product. It has become feasible, through various architectural approaches for building applications, to collaboratively integrate different programming languages within the same project, providing various types of services within a unified framework. As a result, software development has evolved beyond the programmers involved in the project, necessitating the participation of individuals responsible for design, sales, and system maintenance. This includes troubleshooting and managing the system infrastructure itself. Recognizing this, there arises a need to abstract the overall structure of the software to create a more accessible visualization for everyone involved in the software development process. Therefore, with the goal of contributing to the evolution of software production processes and promoting collaboration among as many people as possible in this endeavor, this project aims to create an application capable of obtaining, processing, and metricizing data from an external application, and generating visualizations to represent the relationships and structures within a software project.

Keywords: *Program understanding, program visualization, program metrics, program maintenance.*

LISTA DE FIGURAS

ABREVIATURA

- **APM - Application Performance Monitoring**

SUMÁRIO

1 INTRODUÇÃO

Antes de introduzir os pontos que foram desenvolvidos durante o projeto, se faz necessário apresentar o embasamento teórico abordado ao longo do trabalho, servindo como uma ambientação para os termos e motivações do projeto.

Um dos tópicos centrais tratados neste trabalho é a compreensão de um projeto de software, interligando monitoramento e observabilidade. Porém quais as diferenças entre estes dois termos ?

De forma geral, ambos possuem a mesma finalidade: coletar e armazenar dados de métricas, logs, eventos e rastreamentos. Contudo, como exemplificado no artigo publicado pela Amazon: “Qual é a diferença entre observabilidade e monitoramento?” (AMAZON WEB SERVICES, [2024?]), podemos definir o termo monitoramento como um componente essencial da observabilidade, pois o processo de monitorar um software se baseia na geração de artefatos como: logs, métricas, rastreamentos e eventos. Estes, ao serem armazenados e comparados com resultados antigos, permitem gerar uma visualização do comportamento de determinados componentes presentes em um projeto. Dessa maneira, pode-se utilizar estes artefatos para gerar análises que servirão de base para as investigações feitas acerca dos problemas encontrados no projeto; esta é a função da observabilidade.

Com a contínua evolução da tecnologia, o aumento da complexidade no desenvolvimento de aplicações e as diversas arquiteturas para construir um projeto, tornou-se necessário criar métodos e ferramentas para abstrair esta quantidade de dados e transformar em artefatos que sejam de fácil compreensão para os que estão envolvidos nesta esteira de desenvolvimento. Desta forma, esta monografia surge como uma iniciativa para solucionar este problema que diversos profissionais da área de tecnologia da informação vem enfrentando nos dias atuais.

A motivação surgiu do relato de muitos profissionais que enfrentam dificuldade de compreender e manusear todas as funcionalidades presentes em um projeto com diversos serviços que constantemente interagem entre si, movimentando, tratando e validando dados que serão transportados e utilizados em outro ponto da aplicação. Não obstante a isso, é notório a necessidade de empresas de médio e grande porte criarem representações de mais alto nível de seus softwares para colaboradores que não atuam diretamente na produção e

manutenção do mesmo, como pode ser validado pelo artigo publicado pela IBM em seu artigo publicado sobre: “Qual é o valor da observabilidade para sua organização” (IBM, 2024), em que diz:

> [...]*Por isso, as organizações de TI estão investindo pesadamente em observabilidade, automação de TI e soluções de AIOps. 47% das organizações estão fazendo isso, de acordo com uma pesquisa recente, sendo que 44% citam uma maior adoção de ferramentas de observabilidade e monitoramento.*”

Dado este ponto, é possível validar a necessidade de se produzir um projeto que visa ampliar a efetividade do processo de observabilidade para um software.

Outro tópico que será abordado neste projeto é o conceito de arquitetura de software e como um processo pode ser estruturado para compor o desenvolvimento de um software. Como explicado no livro de Ford e Richards (2020), pode-se citar os três principais modelos existentes no mercado de trabalho:

- **Arquitetura monolítica**, um modelo tradicional de desenvolvimento em que se utiliza um único componente para executar diversas responsabilidades, ou então como é comumente chamado: **regras de negócio**
- **Arquitetura de microsserviços:** diferentemente do modelo tradicional, esta arquitetura busca dividir as regras de negócio de uma aplicação por suas funcionalidades, buscando fornecer um ambiente isolado que possa se comunicar livremente com quaisquer outros módulos que existam no ambiente. Dessa maneira, se torna possível construir serviços que não dependam de uma mesma linguagem de programação, e também não possuam dependências com outros módulos.
- **Arquitetura híbrida** que busca servir como um agregado dos dois outros padrões supracitados, buscando dividir as principais regras de serviços em módulos isolados e independentes e também viabilizar o agrupamento de funcionalidades de diferentes segmentos.

A partir destes pontos, pode-se introduzir o tema central deste projeto, o qual tem como grande desafio a construção de um microsserviço que consiga ser genérico o bastante para ser integrado a qualquer tipo de sistema, sendo indiferente a sua arquitetura. Visando fornecer uma ampla visualização do projeto de maneira simplificada, o projeto deve ser passível de integração para qualquer projeto que necessite de um serviço que colete essas informações e gere uma visualização.

Outro ponto acabou sendo um desafio para o produto foi se diferenciar de produtos já existentes no mercado. Contudo este ponto será melhor destacado no *Capítulo 3* do projeto.

1.2 Detalhamento do problema

Nos dias de hoje, quando avalia-se o processo de construção de um projeto de software, ao analisar o grupo de indivíduos que atuam nesta esteira de produção, é comum encontrar diferentes times de desenvolvedores alocados em um mesmo projeto. Estes, que por sua vez, utilizam diferentes linguagens de programação, como por exemplo: a construção de uma aplicação que possua um módulo de serviço que utilize como linguagem de desenvolvimento o Java para gerenciar e criar suas requisições e para a criação de uma interface visual, utilize o JavaScript para construir um site que consuma as requisições geradas pelo módulo de serviço feito em Java. Dentro deste cenário também é possível elencar colaboradores não ligados ao desenvolvimento em si, como por exemplo: marketing, design, gestão, infraestrutura. Analisando este ponto torna-se necessário fornecer uma visão limpa e concisa do projeto para que todos os envolvidos saibam os pontos principais contidos e as funcionalidades presentes no produto, para assim garantir um maior entendimento do ambiente ao qual estão inseridos.

No entanto, em se tratando de projetos que sejam desenvolvidos com arquitetura voltada para microsserviços ou então arquitetura híbrida contendo microsserviços e estruturas em formato monolítico, a tarefa de mapear as funcionalidades e relacionamentos entre os diversos serviços contidos neste ambiente pode acabar se tornando muito difícil, sendo necessário alocar um número de colaboradores que deverão mapear cada processo de requisição e todas as etapas que ele irá passar. Quando aplicado este mesmo processo de mapeamento para produtos de grande porte essa tarefa pode ser muito longa e difícil, gerando artefatos com erros que resultam em representações incorretos, prejudicando o entendimento geral do produto, sendo necessário retrabalho de alto valor para empresa, por alocar desenvolvedores com alto nível de custo para reavaliar o mapeamento e relacionamento entre os módulos do produto.

A partir desta análise de mercado, buscando diminuir o impacto de tais artefatos, cresce a necessidade de construir projetos focados diretamente na metrificação e mapeamento de softwares, para facilitar o processo de gerar

visualizações de diversos pontos do projeto. No entanto, tal solução deve apresentar um desempenho que não afeta os serviços já existentes do cliente, evitando queda nos níveis de performance do produto, visando apenas servir como um meio de gerar metrificação e melhora no ambiente de trabalho. Este projeto apresenta um produto que será vendido com licença de uso, tendo a manutenção de seus módulos por parte dos desenvolvedores que participaram de sua criação.

Desta maneira, o projeto a ser desenvolvido nesta pesquisa visa servir como solução para diversas empresas, que necessitam a adotar dentro de seus produtos uma solução para os problemas supracitados, visando facilitar o entendimento dos desenvolvedores que buscam conhecer os detalhes e minúcias do projeto, assim como os colaboradores que atuam em outros setores que não possuem contato com o código-fonte mas por participar da esteira de desenvolvimento devem conhecer o produto com o qual atuam.

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, serão apresentados os principais conceitos para desenvolvimento desta monografia, sendo estes responsáveis por servir como base teórica para o desenvolvimento da aplicação.

2.1 Engenharia de Software

A Engenharia de Software pode ser descrita como “uma disciplina que se ocupa do desenvolvimento sistemático, operação e manutenção de software, utilizando métodos e ferramentas que asseguram qualidade, eficiência e funcionalidade” (SOMMERVILLE, 2011). Tal área é responsável por definir desde o início de um projeto de construção de software, com a definição de requisitos, até a entrega do sistema, garantindo qualidade e sustentabilidade durante o ciclo de vida do produto.

2.2 Arquitetura de Software

A arquitetura de software é a disciplina responsável por organizar os pontos fundamentais de um sistema, descrito por seus componentes, a forma como eles interagem e quais serão os princípios de construção de software que serão seguidos durante o desenvolvimento do projeto, sendo descrita por Garlan (1996) da seguinte forma:

> *“[...] As questões estruturais incluem a organização básica e a estrutura global de controle; os protocolos para comunicação, a sincronização e o acesso aos dados; a atribuição das funcionalidades aos elementos de design; a distribuição física; a composição de elementos de design; a característica escalar e o desempenho; e a seleção entre as alternativas de design.[...]”*

Neste projeto, foi utilizado em especial a arquitetura de microsserviços, esta abordagem possui como benefícios a escalabilidade e flexibilidade, pelo fato de possuir um alto nível de desacoplamento este os módulos presentes na aplicação. Contudo, ela também possui alguns desafios, como a necessidade de implementar métodos responsáveis por fazer a comunicação entre os microsserviços e também apresenta como desafio o gerenciamento das dependências utilizadas dentro de cada módulo presente na aplicação. Tal conceito foi especificado pela IBM(2024), “[...] *é uma abordagem de arquitetura nativa da nuvem na qual uma aplicação é*

composta por muitos componentes ou serviços menores, que são acoplados e implementados de forma independente, comunicando-se por APIs".

2.3 Visualização de software

A visualização de software pode ser definida como uma técnica que utiliza representações visuais para facilitar o entendimento e análise de sistemas complexos. Segundo Diehl, "*a visualização de software abrange o desenvolvimento e avaliação de métodos gráficos para representar diferentes aspectos do software, incluindo sua estrutura, sua execução e suas possíveis evoluções…*"

Tal técnica é fundamental para ambientes com alto nível complexidade do código e com arquiteturas com difícil entendimento.

2.4 Métricas de desempenho

Métricas de desempenho são indicadores quantitativos utilizados para avaliar a eficiência e a eficácia de um sistema. São comumente utilizados para mensurar tempo de resposta, taxa de transferência e uso de recursos dentro de um sistema.

No contexto de microsserviços, métricas como latência, disponibilidade e taxa de erros são fundamentais para avaliar o desempenho de serviços individuais e da aplicação como um todo (NEW RELIC, 2024). Dessa forma, ferramentas que fazem a coleta e interpretação dessas métricas, são de extrema importância para a organização.

3 SOLUÇÕES EXISTENTES

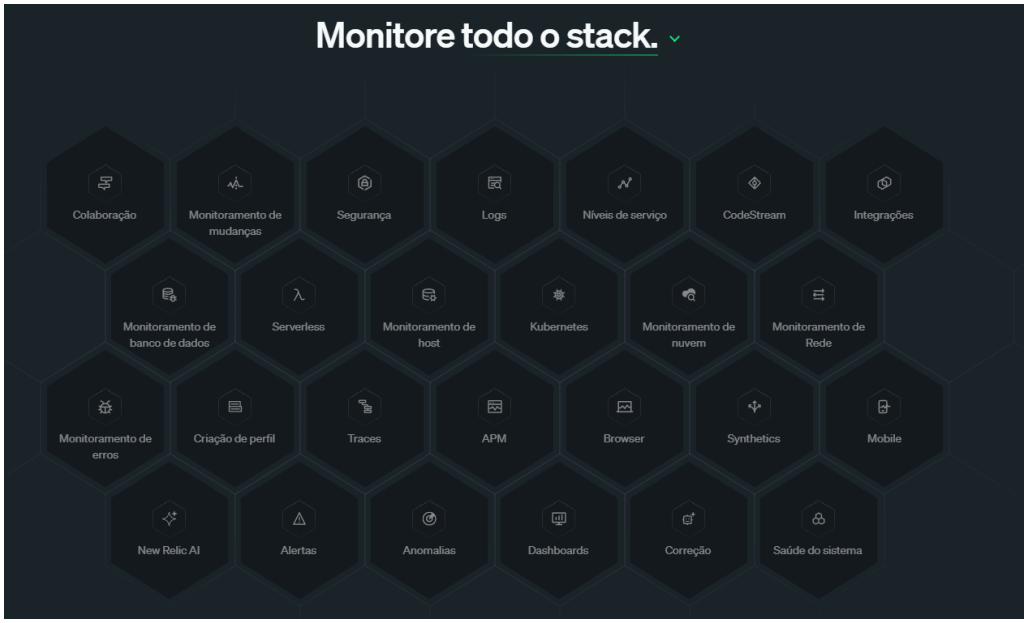
Neste capítulo será abordado sistemas já presentes no mercado que apresentam resoluções para o problema de metrificação e geração de visualização, eles são: o New Relic uma plataforma de serviços pagos que apresenta diversas soluções de software incluindo metrificação e rastreamento distribuído, e o Jaeger uma plataforma *open source* focada apenas em rastreamento distribuído.

3.1 New Relic

O New Relic, se trata de uma aplicação *full-stack* focada no monitoramento de performance atendendo pela sigla: *APM (Application Performance Monitoring)*. Fornecendo ao usuário uma visão completa de todos os componentes presentes em seu projeto, sendo possível atender quaisquer as necessidades de monitoramento e observabilidade que o projeto precise.

A plataforma também disponibiliza uma dashboard que pode ser personalizada pelo usuário, sendo possível extrair informações de fontes terceiras, e alterado para melhorar o desempenho do colaborador, fornecendo informações precisas sobre diversos pontos do projetos, como por exemplo: log de erros em serviços, requisições feitas entre módulos, análise de distribuição de erros, entre outros fatores disponibilizados na Figura 1, adjunto a isso é possível visualizar um exemplo da dashboard criada na Figura 2.

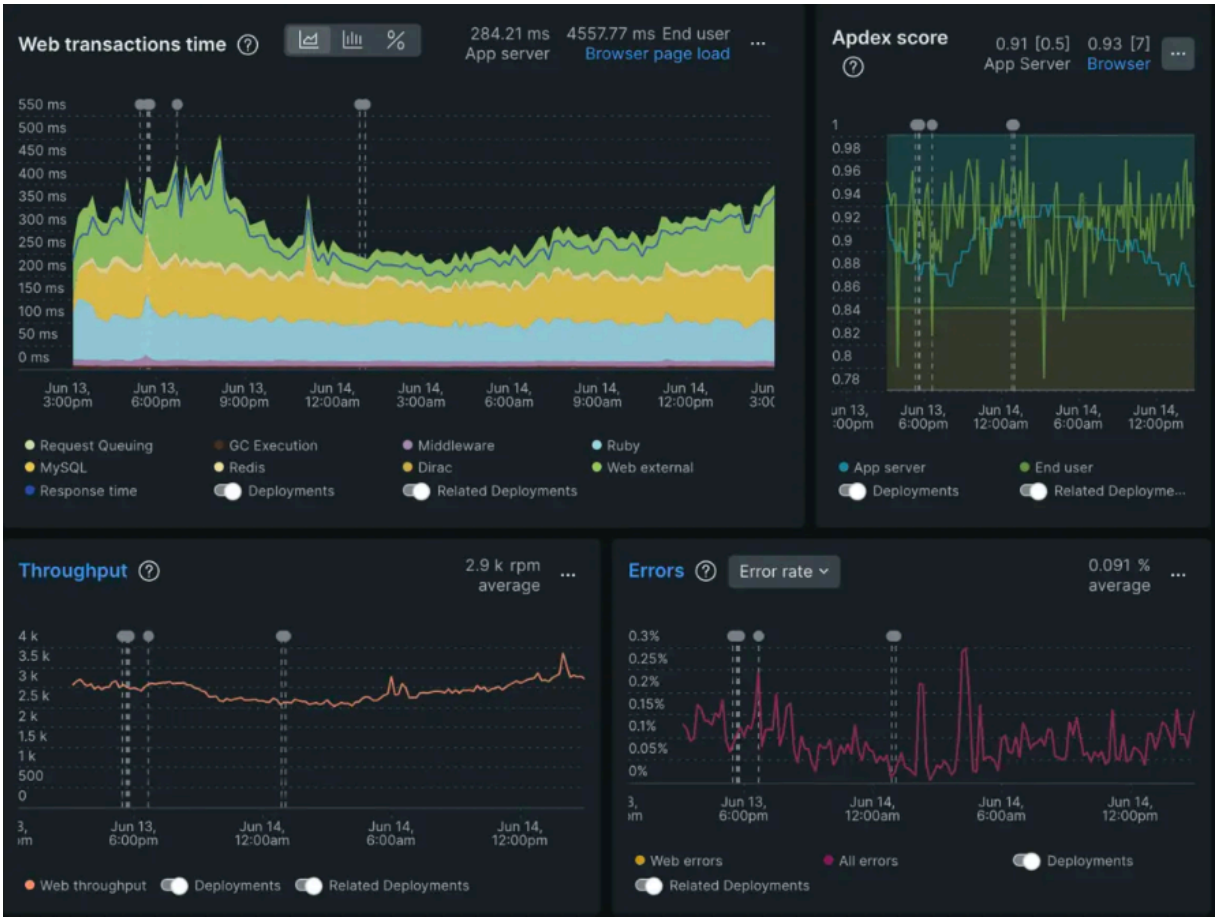
Figura 1 - Funcionalidades presentes na plataforma (New Relic).



Fonte: New Relic, 2024.

Um exemplo da dashboard é mostrado na Figura 2.

Figura 2 - Exemplo de dashboard criada pela plataforma.



Fonte: New Relic, 2024.

Com relação aos investimentos necessários para uma empresa adquirir os serviços da plataforma New Relic, a plataforma fornece uma quantidade de dados grátis por mês, dados esses que são inseridos na plataforma para gerar os resultados de monitoramento, e também oferece três padrões diferentes de usuários para configuração de ambientes, configurado de acordo com as necessidades da empresa contratante para com a plataforma.

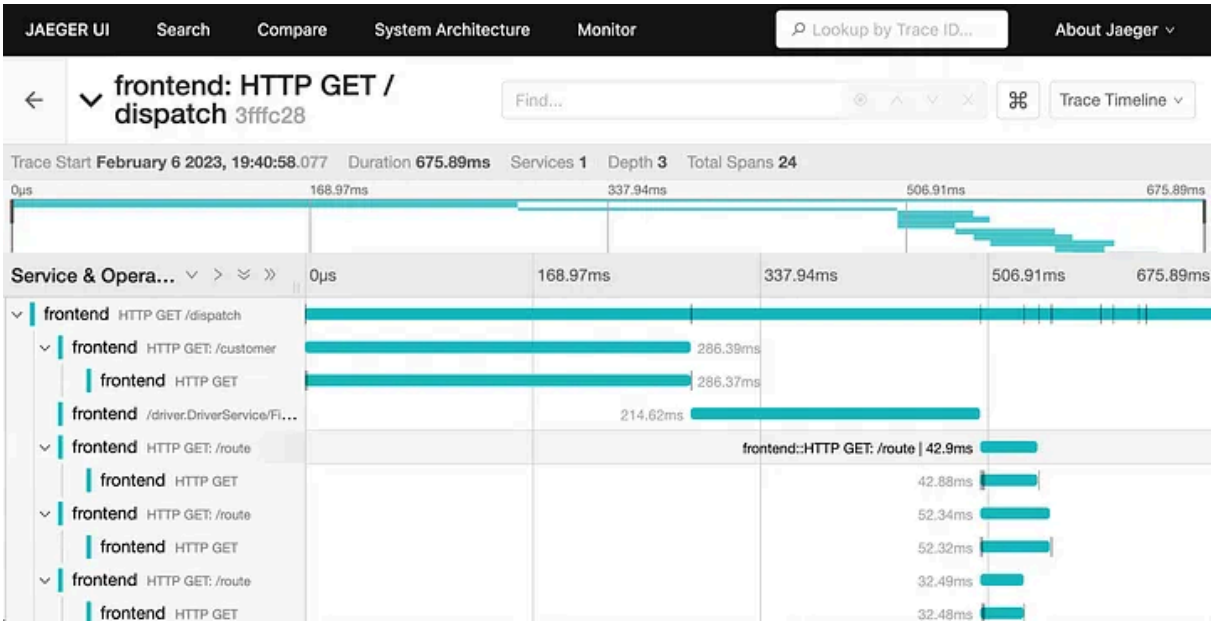
Contudo, a plataforma New Relic possui pontos negativos muito claros, que este trabalho visa solucionar, como por exemplo, a alta complexidade do dashboard, o qual necessita de um treinamento específico para entender suas funcionalidades e conseguir usufruir completamente delas. Outro ponto negativo que pode ser elencado é a falta de uma representação de alto nível, visando informar colaboradores que não possuem todo o conhecimento necessário para avaliar o

conteúdo contido nos dashboards e consequentemente gerando um problema de entendimento dos mesmos.

3.2 Jaeger

A segunda solução existente no mercado apresentada neste capítulo é a plataforma *open source* “Jaeger”, focada exclusivamente em rastreamento distribuído, tema que será abordado e explicado no Capítulo 4. A plataforma possui uma interface gráfica que disponibiliza as requisições presentes no projeto e todo o mapeamento de rotas, fornecendo uma visão clara do caminho percorrido por todo o projeto. É possível visualizar um exemplo da interface gráfica na Figura 3.

Figura 3 - Exemplo da interface gráfica do Jaeger.



Fonte: Jaeger, 2023.

Esta opção, atua de forma colaborativa, ou seja, ela é mantida por meio de colaboradores que buscam melhorar e corrigir problemas que surjam durante o processo de desenvolvimento de código aberto.

A partir deste ponto, pode-se elencar o principal problema desta opção. Considerando um cenário empresarial ocorrerão casos que será necessário solicitar suporte para correção de problemas na plataforma, contudo por não haver uma empresa direta que atua no projeto buscando sempre fornecer um produto sem bugs para seu cliente, a empresa teria que recorrer aos seus próprios desenvolvedores

para resolver o problema, ou então esperar uma liberação feita por um desenvolvedor colaborador da plataforma.

Além deste fator, o Jaeger apresenta o mesmo problema listado na plataforma New Relic, a ausência de uma visão de alto nível que seja de fácil entendimento para colaboradores que não atuem com desenvolvimento.

3.3 Considerações

Tendo em vista os pontos fortes destes produtos já existentes no mercado, este projeto fornece uma solução focada em um resultado que possa ser apresentado para todos os colaboradores de uma empresa e seja de fácil entendimento, fornecendo as principais informações do projeto, como por exemplo: quais são os componentes de microsserviços que estão alocados na arquitetura do projeto, quais são as requisições presentes neste serviços, e por fim traçar uma representação visual sobre o trajeto da requisição entre os microsserviços.

Após analisar as principais opções de mercado, elencando tanto um projeto *open-source* quanto um projeto comercial foi possível notar a necessidade do projeto possuir visões direcionadas para diferentes perfis de colaboradores, podendo agregar valor para vários setores de uma mesma empresa.

Portanto, após os pontos levantados, este projeto corrige os pontos negativos, buscando fornecer uma plataforma que consiga metrificar um projeto de software e gerar uma visualização acessível para diversos públicos dentro de uma mesma empresa, sendo uma plataforma de fácil acesso para todos os colaboradores.

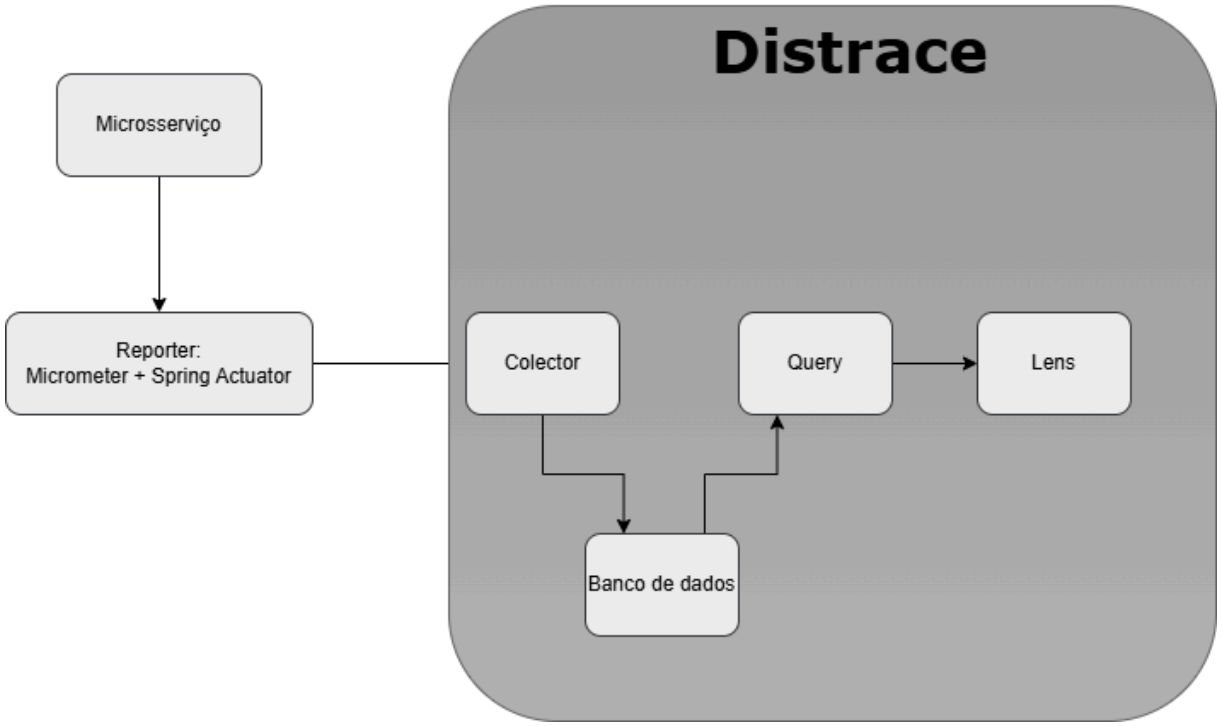
4. DESCRIÇÃO DO SOFTWARE

Este projeto tem como função construir uma aplicação chamada Distrace, que atua como uma extensão visual para projetos de software, fornecendo uma interface gráfica interativa que representará os módulos de microsserviços contidos dentro do projeto. O Distrace utiliza bibliotecas para capturar as métricas contidas no projeto e construir representações visuais de tais informações. O Distrace é composto pelos seguintes módulos:

- ***Colector***: responsável por coletar essas metrificações disponibilizadas, como por exemplo: quais são as requisições disponíveis, se elas estão funcionando e por fim, armazenar em um banco de dados para posteriormente ser repassadas para o próximo módulo de tratamento.
- **Query**: é responsável por fazer as requisições no banco de dados e enviá-las diretamente para a interface gráfica que irá gerar as representações visuais.
- **Lens**: módulo responsável por gerar o conteúdo gráfico presente na aplicação, sendo a aplicação front-end do projeto.

A Figura 4 mostra um exemplo do fluxo de sequência do projeto Distrace

Figura 4 - Visualização do projeto.



Fonte: Autoria própria.

O projeto conta com a biblioteca Open Telemetry para fornecer e exportar dados de telemetria, possibilitando rastrear os métodos presentes nos componentes, quais as requisições disponíveis, os índices de performance de cada requisição e a disponibilidade de tais serviços.

Para o rastreamento das requisições, é utilizado a extensão pertencente à versão 3.2.4 conhecida como Distributed tracing, que possui as seguintes bibliotecas: Micrometer e Spring actuator responsáveis por mapear o ambiente de teste e enviar as métricas para o ambiente onde esses dados serão armazenados e tratado pelo módulo desenvolvido neste projeto.

4.1 Microsserviços desenvolvidos

Os microsserviços que servirão como base para a análise do módulo de criação de interface gráfica, foco deste trabalho. Para facilitar essa análise, construiu-se ambientes de fácil compreensão e com configurações simples, simulando cenários de requisições que interagem entre esses módulos.

Dessa forma, os módulos são de fácil compreensão, com requisições que interligam os projetos. Isso permite que no módulo de criação de interface gráfica, desenvolver estruturas que representem as comunicações entre esses microsserviços. Para a construção destes microsserviços foram utilizados as seguintes tecnologias:

- *Spring Actuator:* Utilizado para monitorar e gerenciar os serviços do projeto possibilitando acessar as métricas de desempenho, saúde e também especificar quais as requisições presentes dentro do microsserviço em questão (Spring Framework, 2024).
- *Micrometer*: Também se trata de uma ferramenta de gerenciamento dos serviços, porém é especializada em fazer a ligação entre os dados e o sistema de monitoramento. (Micrometer, 2024).
- *Zipkin:* Ferramenta responsável por fazer o rastreamento distribuído responsável por coletar os dados de latência, e também fornecer o fluxo das requisições. (Zipkin, 2024).
- *Lombok:* Tecnologia responsável por automatizar a geração de código, visando reduzir a verbosidade do projeto, e facilitar o entendimento do código. (Project Lombok, 2024).

4.2 Módulo do Distrace

A segunda parte deste projeto envolve o desenvolvimento de uma customização no Zipkin, tecnologia que coleta dados de latência em sistemas e mapeia o fluxo das requisições realizadas. Essa customização é utilizada para criar um sistema gráfico que ofereça uma representação visual dos microsserviços do projeto, aprimorando o monitoramento e a análise das interações entre eles. Com isso, obtendo uma visão mais detalhada do desempenho e identificar possíveis pontos de melhoria.

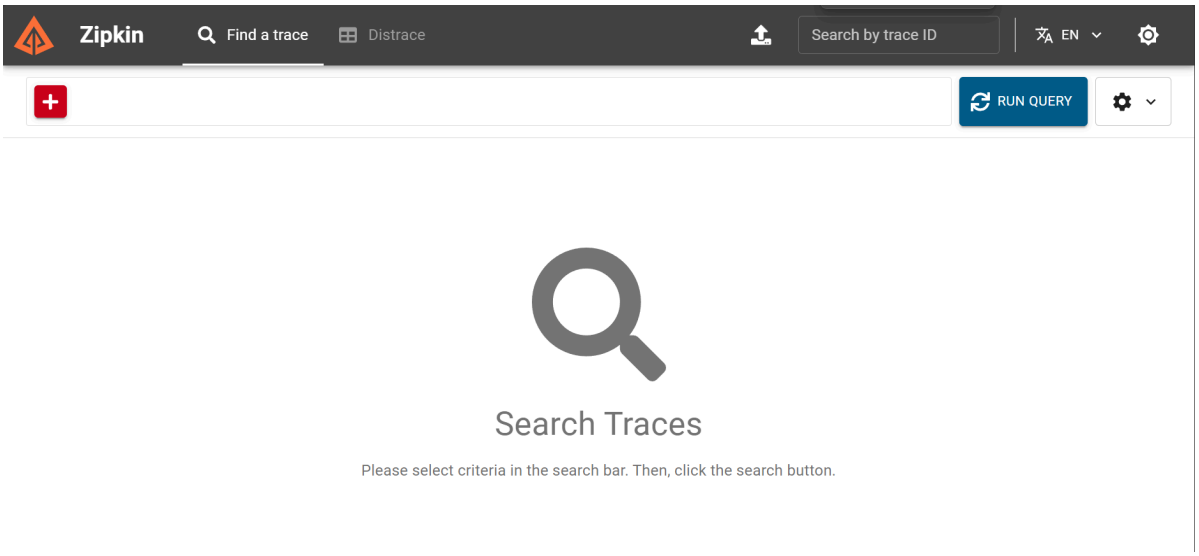
Pensando na comunicação entre os serviços, foi utilizado o protocolo REST, que é altamente recomendado para arquiteturas de sistemas distribuídos. Ele permite que os microsserviços se comuniquem utilizando operações baseadas em

HTTP, como **GET**, **POST**, **PUT** e **DELETE**, que correspondem a ações como leitura, criação, atualização e remoção de recursos.

No contexto deste projeto, o uso do protocolo REST é essencial para possibilitar a integração e o rastreamento das interações entre os microsserviços. Essa comunicação, monitorada pelo Zipkin, fornece os dados necessários para criar uma visão gráfica do sistema, permitindo identificar gargalos, pontos de latência elevada ou falhas no fluxo de requisições. Dessa forma, a customização desenvolvida visa otimizar tanto a análise quanto a resolução de problemas, promovendo melhorias contínuas no desempenho do sistema como um todo.

Para construir essa customização, foram utilizadas as tecnologias já presentes no Zipkin, com alterações no serviço chamado "Zipkin-lens", que é responsável pela interface gráfica. Na Figura 5 é possível ver um exemplo da interface gráfica nativa da plataforma Zipkin.

Figura 5 - Página Web do projeto.



Fonte: Autoria própria.

4.3 Tecnologias utilizadas

As tecnologias utilizadas no projeto, são tratadas em tópicos para facilitar a leitura e entendimento dos módulos desenvolvidos neste trabalho.

Linguagem de programação

- Foi escolhido como linguagem e desenvolvimento o Java em sua versão 17 para ser possível utilizar as funcionalidades mais recentes presentes na plataforma e ser compatível com as frameworks. De forma geral, esta foi a linguagem utilizada em todos os escopos do projeto.

Ambiente de teste

- O ambiente de teste foi construído inteiramente na linguagem Java versão 17, utilizando a framework Spring Boot para criação de um projeto que servindo como teste de mapeamento dos microsserviços que lá existem
- Para gerar as métricas deste ambiente e enviar para o serviço de "*Coleta"* é utilizado as frameworks Micrometer e Spring actuator, que respectivamente servem para geração e coleta de métricas presentes no ambiente, e fornecer métodos em tempo de execução para acessar tais informações e a framework SLF4J para geração de logs padronizados.
- E por fim, para enviar essas informações também é utilizado a framework Spring Micrometer, que é responsável por definir os roteamentos de requisições entre os serviços.

Colector

- O serviço de coleta de dados foi desenvolvido utilizando Java. Logo, para simular um ambiente que possui requisições entre microsserviços foi utilizado a biblioteca: RestTemplate para criar requisições entre os módulos, sendo utilizado o padrão de desenvolvimento levantado pelo projeto *open source* ZipKin, contudo sendo alterado para a proposta levantada por este trabalho
- Foi utilizado o framework JPA para armazenar os dados recolhidos, tornando viável armazenar o conteúdo sobre histórico do sistema ao qual ele está ligado.

Query

- O serviço responsável por administrar as requisições em banco de dados, para alimentar o ambiente é responsável por gerar a visualização, utilizando a framework do Java JPA e para enviar as requisições para o ambiente web faz uso da framework Spring Boot e Web Spring

Lens:

- A visualização utiliza JavaScript em seu desenvolvimento com a framework React.JS para criação de componentes para gerar as visualizações gráficas das métricas recebidas.
- Para a criação das representações visuais em formato de grafo, foi utilizado o framework *Vizceral*, com destaque para sua extensão *VizceralWrapper*. Essa extensão é responsável por monitorar sistemas distribuídos, simplificando a integração, personalização e uso de visualizações interativas, permitindo a análise e representação de tráfego de rede e fluxos de dados.

5. TESTES DE VALIDAÇÃO

Neste projeto, os testes para validação do funcionamento dos componentes foram desenvolvidos utilizando o framework JUnit, amplamente empregado na linguagem Java para a criação de testes unitários.

O principal objetivo desses testes é assegurar que os resultados sejam adequados, possibilitando a geração precisa do conteúdo gráfico baseado nos dados fornecidos pelo back-end. A Figura 6 mostra um exemplo do código de testes da aplicação.

Figura 6 - Teste da estrutura de dados grafo.

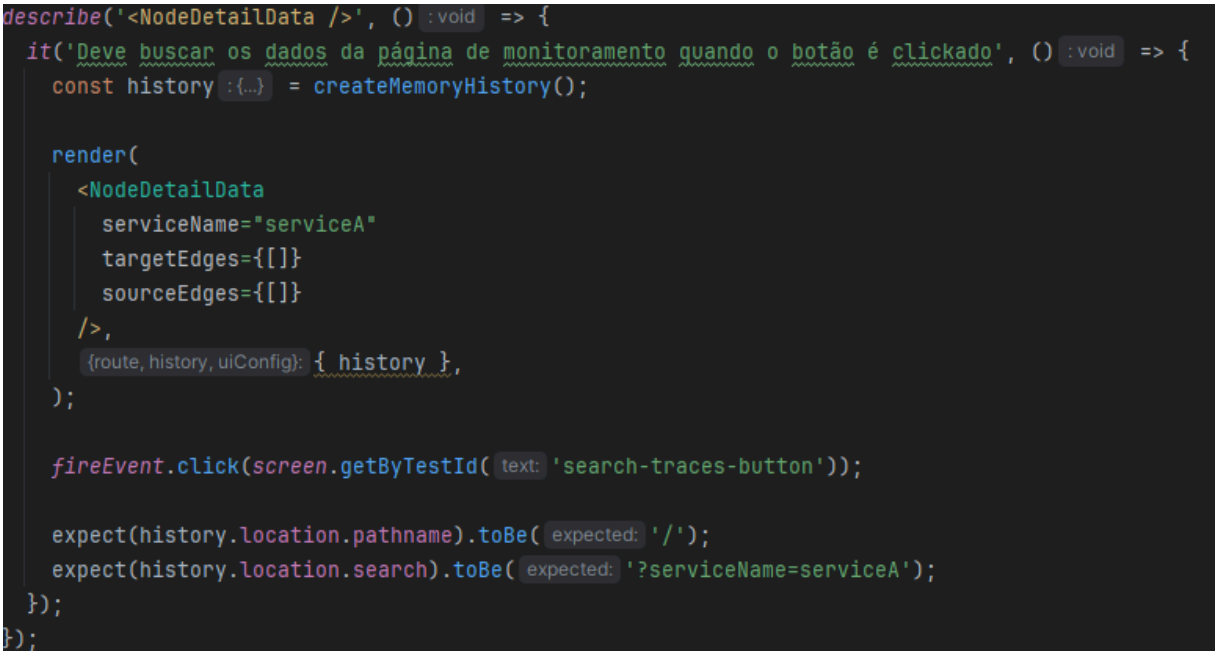
```
it('getNodesAndEdges', () : void => {
  const dependencies : [{parent: string, callCount: n... = [
    {parent: 'A',
      child: 'B',
      callCount: 10,
      errorCount: 3,
    },{
      parent: 'B',
      child: 'E',
      callCount: 20,
      errorCount: 7,
    },{
      parent: 'A',
      child: 'C',
      callCount: 12,
      errorCount: 1,
    },{
      parent: 'C',
      child: 'D',
      callCount: 7,
      errorCount: 0,
    },{
      parent: 'D',
      child: 'E',
      callCount: 3,
      errorCount: 1,
    },
  ];
  const { nodes :({...})[] , edges : Edge[] } = getNodesAndEdges(dependencies);
  const nodeNames : string[] = nodes.map((node : {name: string} ) : string => node.name);
  expect( val: ['A', 'B', 'C', 'D', 'E']).toEqual(expect.arrayContaining(nodeNames));
  expect(edges[0]).toEqual( expected: {
    source: 'A',
    target: 'B',
    metrics: {
      normal: 10,
      danger: 3,
    },});});
```

Fonte: Autoria própria.

Este teste foca em verificar a funcionalidade de criação de *nodes*, que representam o relacionamento entre microsserviços. Nesse contexto, o *parent* é o microsserviço que realizou a requisição anterior, enquanto o *child* é o serviço que foi chamado posteriormente na ordem de execução.

Outro teste que pode ser exemplificado foi focado na funcionalidade de requisitar os dados para a tela de monitoramento, podendo ser visto na Figura 7.

Figura 7 - Teste de direcionamento de dados



Fonte: Autoria própria.

O teste tem como objetivo verificar se ao pressionar o botão de busca os dados adquiridos na tela de monitoramento são corretamente direcionados para a aba de que irá gerar a visualização.

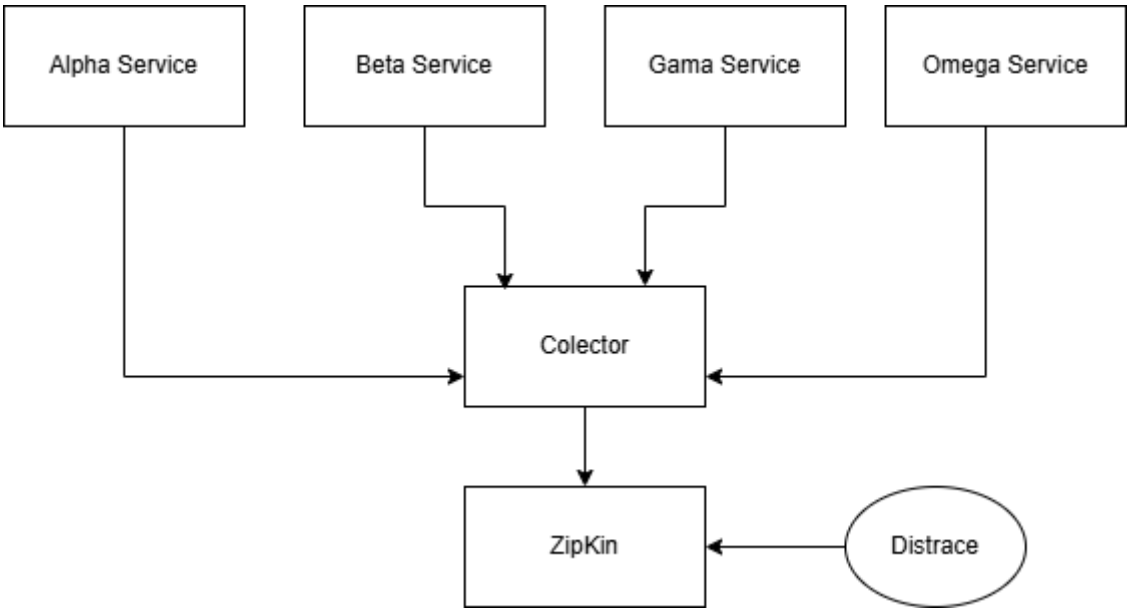
6. IMPLEMENTAÇÃO E USO DO DISTRACE

Este capítulo tem como objetivo apresentar as principais funcionalidades do projeto, começando pelo ambiente que simula as requisições. Em seguida, é detalhado os módulos responsáveis pelo tratamento dos dados e pela criação das representações visuais.

6.1 Estrutura do projeto

Para que seja possível aplicar a solução apresentada pelo projeto Distrace se fez necessário a construção de um sistema que utilize de um conjunto de microsserviços para servir como simulação. Dessa maneira, através da troca de dados entre tais serviços foi possível mensurar e representar a comunicação entre eles para gerar uma visualização do projeto como um todo. É possível visualizar o diagrama que faz referência à arquitetura do projeto na Figura 8.

Figura 8 - Representação do uso do Distrace



Fonte: Autoria própria.

Os microsserviços apresentados na Figura 8 são compostos por uma função responsável por disparar uma requisição que será catalogada pelo sistema Colector e enviada para o sistema *Zipkin* que com a adição do *Distrace* gera a representação gráfica da comunicação entre os microsserviços.

Buscando uma melhor compreensão e integração do projeto com o sistema de observabilidade todo o sistema foi desenvolvido na linguagem Java.

A IDE utilizada foi o IntelliJ por possuir uma maior gama de ferramentas que facilitam no processo de testar e desenvolver os microsserviços, além de facilitar a integração com bases de dados e containers em docker. Todos os microsserviços foram desenvolvidos utilizando os frameworks Micrometer e ZipKin para capturar os dados de usabilidade do sistema e enviar para o serviço que gera a representação gráfica dos pontos levantados.

6.2 Distrace

Para esta parte do projeto, utilizou-se o framework Zipkin como base devido à sua capacidade de coletar e processar dados de forma rápida e eficiente. Para isso, é necessário configurar um módulo de mensageria, responsável por transmitir os dados para o serviço, onde serão tratados.

Com essa estrutura, foi criada uma customização que, a partir dos dados recebidos, gera representações gráficas dos serviços envolvidos em cada requisição do projeto, facilitando a visualização e o monitoramento das interações entre os microsserviços. A Figura 9 apresenta a tela de requisições do projeto.

Figura 9 - Requisições do projeto

Fonte: Autoria própria.

6.3 Microsserviços

Para a implementação dos microsserviços foi utilizado as tecnologias previstas na Seção 4.1, onde foi especificado quais seriam os aspectos para a construção dos módulos presentes no ambiente de teste.

No processo de construção dos serviços foi necessária a utilização da tecnologia de automação e compilação de código para projetos em Java, conhecida como Maven, para que, dessa maneira, fosse possível organizar as importações e os processos necessários para a manutenção e o build do projeto.

Os microsserviços presentes no projeto foram desenvolvidos para serem facilmente reconhecidos pelo sistema de rastreamento de requisições e também para possuírem um fácil entendimento de seu funcionamento, visando que todos os usuários do sistema para fins de estudo possam entender como cada requisição

utilizada funciona. A Figura 10 apresenta o código da implementação da comunicação entre os módulos.

Figura 10 - Código das requisições.

```java
@RestController
@Slf4j
public class AlphaController {

    final RestTemplate restTemplate;

    public AlphaController(RestTemplate restTemplate){
        this.restTemplate = restTemplate;
    }

    @GetMapping("/alpha")
    @Observed(
            name = "user.name",
            contextualName = "alfa-->beta",
            lowCardinalityKeyValues = {"userType", "userType2"}
    )
     public String comunicacaoModulos(){
        log.info("Serviço Alpha foi chamado");
        log.info("Comunicação de Alpha para Gama");
        ResponseEntity<String> response = restTemplate.exchange(
                url: "http://localhost:6060/beta-svc/beta",
                HttpMethod.GET,
                requestEntity: null,
                String.class
        );

        return "Comunicação entre os serviços(Alpha): "+response.getBody();
    }

}
```

Fonte: Autoria própria.

A estrutura do microsserviço foi desenvolvido da seguinte forma:

- A anotação @RestController indica que a classe é um controlador REST, o que significa que os métodos dessa classe retornarão dados diretamente como resposta HTTP, geralmente em formato JSON. Mais adiante no código, a anotação @GetMapping é usada para especificar o verbo HTTP GET, que atua como um gatilho para que o método associado seja executado sempre que uma requisição GET for realizada no endpoint definido.
- A próxima configuração @Slf4j se trata de uma anotação responsável por gerar logs de forma automática

- O Objeto *RestTemplate* é utilizado para fazer chamadas HTTP a outros serviços, sendo este o responsável pela comunicação entre os serviços.
- A anotação @Observed é utilizada mais adiante no código para rastrear e monitorar métricas da requisição à qual está associada. Essa anotação é personalizável, permitindo configurar diferentes parâmetros para monitoramento. No exemplo foram aplicadas as seguintes configurações:
    - *name = "user.name":* usado para identificar o usuário que está fazendo a requisição.
    - *contextualName = "alfa→beta":* define o contexto do monitoramento, indicando que a comunicação ocorre entre os microsserviços Alpha e Beta.
    - *lowCardinalityKeyValues = {"userType", "userType2"}:* adiciona informações de baixa cardinalidade para o log, indicando o tipo de usuário e o fluxo de comunicação. Nesse caso, *userType* representa o módulo Alpha e *userType2* representa o módulo Beta.
- Sendo assim, o método *comunicacaoModulos* é o responsável pela comunicação entre os microsserviços através do uso da função: *restTemplate.exchange* que realiza uma chamada do verbo *GET* ao endpoint configurado no microsserviço beta: "http://localhost:6060/beta-svc/beta" obtendo como resposta uma string.

Esta classe é responsável por configurar a anotação utilizada no método *comunicacaoModulos*, possuindo os seguintes aspectos apresentados na Figura 11.

Figura 11 - Código de configuração do observador

```
@Configuration(proxyBeanMethods = false)
public class ObserverConfig {

    @Bean
    ObservedAspect observedAspect(ObservationRegistry observationRegistry){
        return new ObservedAspect(observationRegistry);
    }

}
```

Fonte: Autoria própria.

- A anotação *@Configuration(proxyBeansMethods = false)*: é utilizada para melhorar o desempenho da aplicação evitando problemas com proxy.
- Por fim, o método *observedAspect(ObservationRegistry observationRegistry)*: retorna uma instância do objeto *observedAspect*, utilizando o parâmetro *observationRegistry*. Esses objetos são responsáveis por integrar o sistema de monitoramento nos módulos de microsserviços.

Outro ponto importante para o funcionamento do ambiente dos microsserviços foi a configuração de tais módulos de microsserviço. Para isso foi necessário a definição de parâmetros dentro do arquivo de application.yaml visando a compatibilidade e a possibilidade do rastreio das requisições. Esses pontos podem ser vistos na Figura 12.

Figura 12 - Configuração do microsserviço

```yaml
server:
  servlet:
    context-path: /alpha-svc
  port: 7070

spring:
  application:
    name: "Alpha"

logging:
  pattern:
    level: logging.pattern.level=%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]
management:
  tracing:
    sampling:
      probability: 1.0
  endpoints:
    web:
      exposure:
        include: prometheus
  metrics:
    distribution:
      percentiles-histogram:
        http:
          server:
            requests: true
```

Fonte: Autoria própria.

É possível estruturar os parâmetros mostrados na Figura 12 da seguinte maneira:

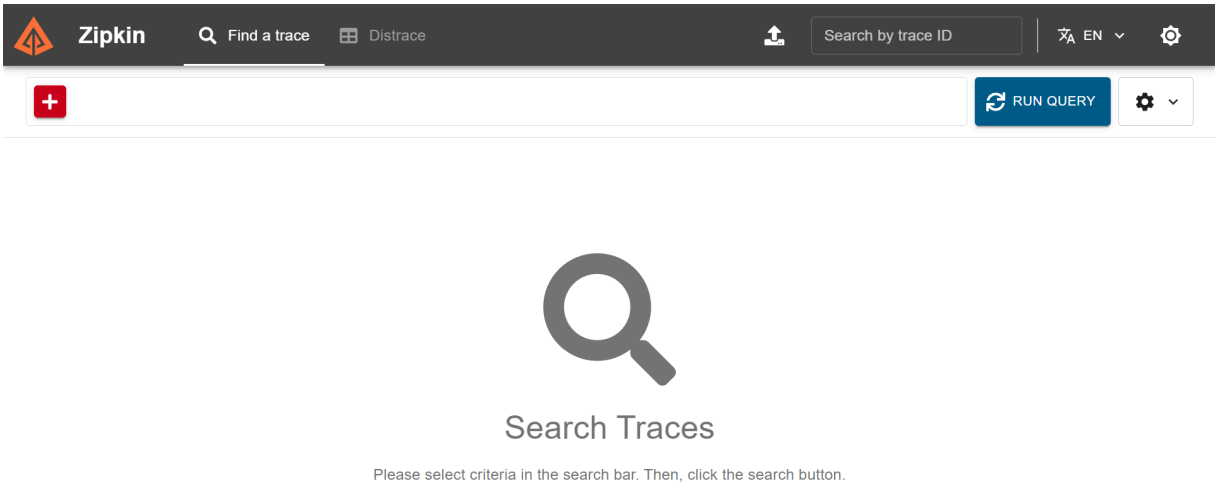
- *Server*: Define as configurações do servidor.
    - Context-path: Representa o caminho da requisição.
    - Port: Corresponde à porta do servidor.
- Spring: Responsável pelas configurações específicas do spring

    - Application: Indica que as configurações a seguir são relacionadas à aplicação.
    - name: Nome da aplicação
- Logging: Configurações de log.
    - Pattern: Define um padrão de log.
    - Level: Especifica o nível de log e gera um padrão de texto a ser exibido através das variáveis de ambiente utilizadas.
- Management: Responsável pelas configurações de gerenciamento e monitoramento.
    - Tracing: Configura o rastreamento.
    - Sampling: Define a probabilidade de amostragem das requisições
    - Probability: Probabilidade de amostragem, informando qual a porcentagem de requisições serão analisadas.
- Endpoints: Configura os endpoints de monitoramento.
    - Web: Configurações de exposição dos endpoints
    - Exposure: Quais serão os endpoints expostos.
    - Include: prometheus: Exposição do endpoint do Prometheus para coletar métricas.
- Metrics: Configurações específicas para métricas
    - Distribution: Distribuição das métricas
    - Percentiles-histogram: Configuração de histograma
    - Http: métricas Http
    - server: Métricas relacionadas ao servidor
    - requests: true: Ativa a coleta de métricas para requisições HTTP do servidor.

6.4 Tela de requisições

Esta seção irá tratar a tela de requisição do projeto sendo possível visualizar uma representação da tela na Figura 13, que é responsável por exibir todo o conteúdo do rastreamento das requisições que foram catalogadas a partir das configurações mencionadas na Seção 7.3.

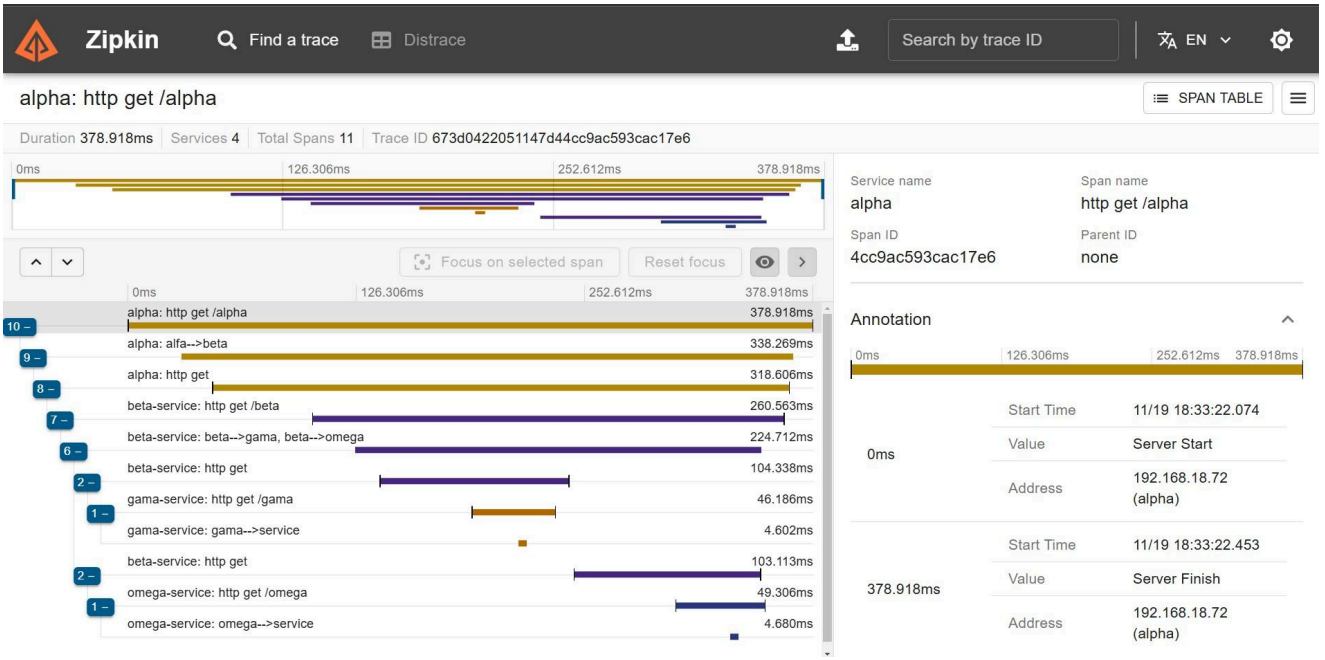
Figura 13 - Tela de requisição sem conteúdo.



Fonte: Autoria própria.

Ao acessar esta tela, é necessário que o usuário clique no botão “Run query” para carregar o conteúdo registrado no banco de dados, exibindo assim as requisições realizadas durante a execução da aplicação. Após o carregamento das informações, o usuário pode habilitar uma visualização mais detalhada de cada requisição, permitindo visualizar a ordem de execução, o tempo necessário para a conclusão da requisição e os detalhes definidos pelas configurações feitas no back-end do projeto, a Figura 14 faz referência a tela descrita neste parágrafo.

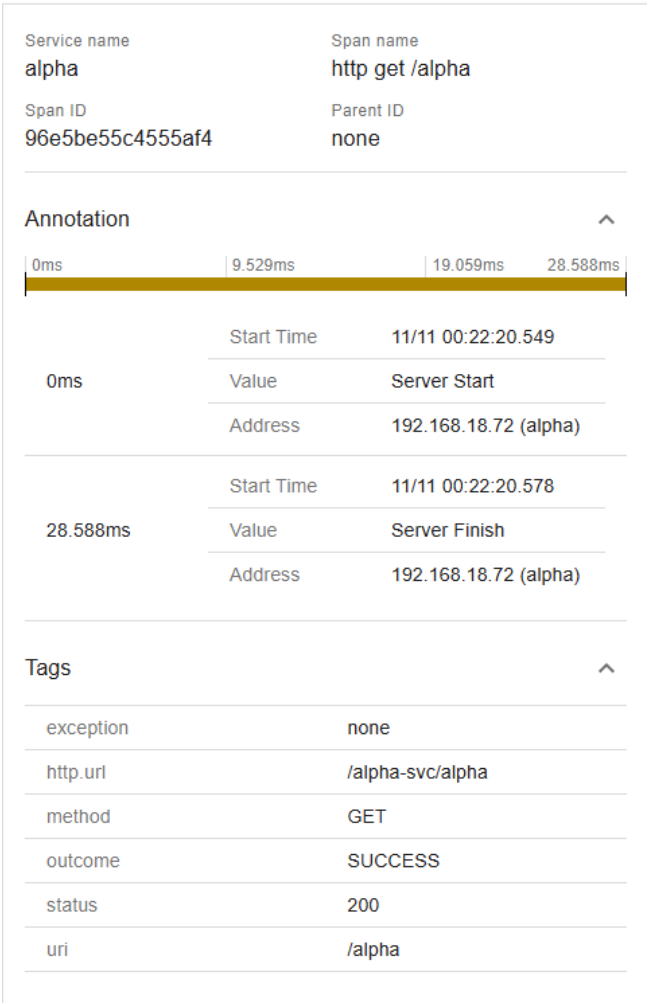
Figura 14 - Tela de requisições



Fonte: Autoria própria.

Ao analisar a aba de detalhamento da requisição é possível encontrar informações como o horário em que a requisição foi iniciada e finalizada, além do endereço ao qual ela pertence. Na aba Tags, estão disponíveis os dados customizados nos arquivos de configuração, incluindo o cliente que fez a requisição, possíveis exceções, a URL de disparo, o método utilizado, o resultado da requisição com seu respectivo status e, por fim a URI um identificador que neste caso indica o caminho específico no servidor onde o recurso ou serviço está localizado, facilitando o roteamento e o acesso aos dados desejados. A Figura 15 faz referência a esta aba.

Figura 15 - Especificação da requisição

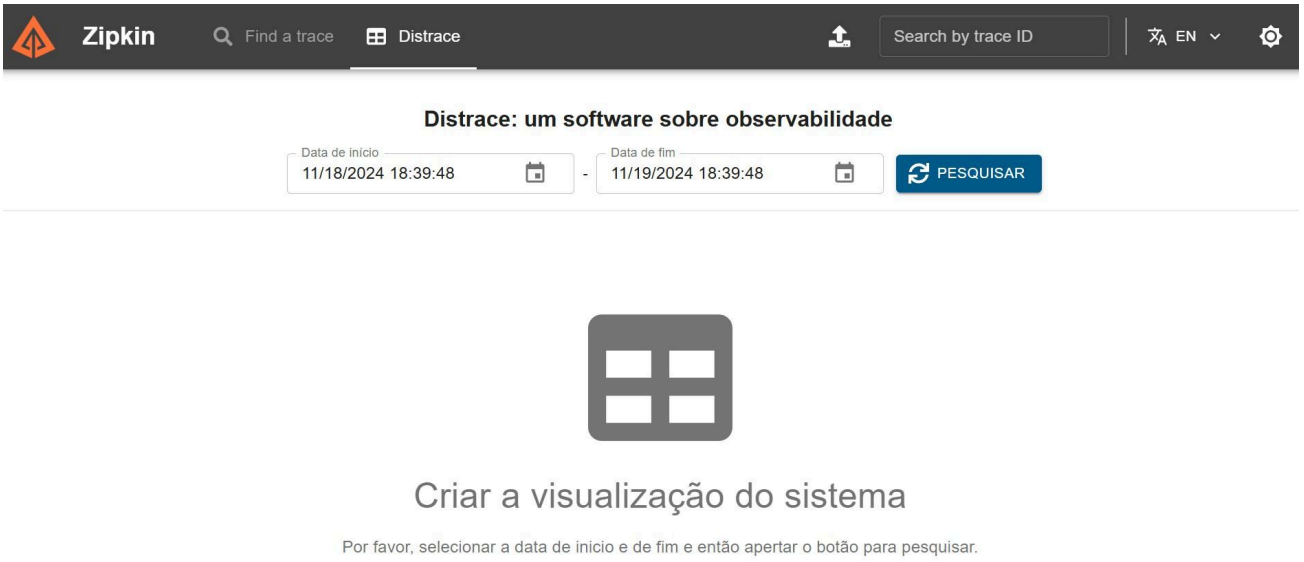


Fonte: Autoria própria.

6.5 Tela de dependências

Outro ponto abordado neste trabalho é a tela de dependências, responsável por gerar a representação gráfica dos microsserviços presentes no projeto e fornecer informações sobre as interações entre eles. Essa tela utiliza os dados coletados pela tela de requisição para gerar as representações dos microsserviços. Esses dados são obtidos por meio das anotações inseridas no código-fonte do projeto, indicando a cardinalidade das requisições, permitindo a geração do grafo de interações dos microsserviços. A Figura 16 faz referência à tela mencionada.

Figura 16 - Tela do Distrace sem conteúdo carregado

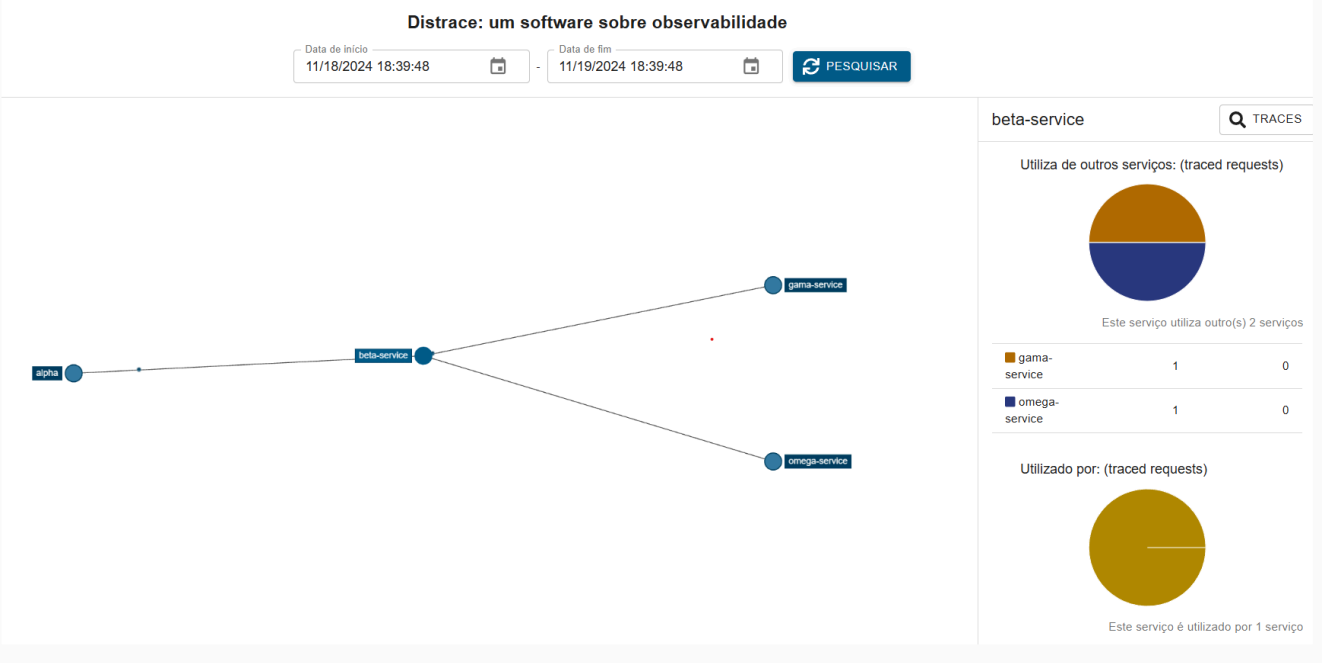


Fonte: Autoria própria.

Na tela da Figura 16 é necessário que o usuário selecione o intervalo de tempo em que o sistema deve fazer a procura das requisições e selecione o botão para que gere a visualização.

Após a pesquisa em um intervalo de tempo específico, é gerado um grafo que informa quais microsserviços interagiram durante esse período, sendo possível visualizar tal funcionamento na Figura 17. Nesse grafo, cada nó representa um botão interativo que, ao ser selecionado, abre um painel com informações gráficas sobre as relações que originaram a comunicação com aquele módulo, bem como as interações que se originaram a partir desse microsserviço.

Figura 17 - Tela do Distrace com conteúdo carregado.



Fonte: Autoria própria.

Para a criação da representação gráfica, baseada em grafos para simbolizar as relações entre os microsserviços, utilizou-se a linguagem TypeScript. Essa escolha permite criar entidades com propriedades semelhantes às disponíveis na linguagem Java, empregada no back-end da aplicação. Dessa forma, foi desenvolvida a representação dos ***Nodes***, que possuem os seguintes atributos:

- **Parent**: Representa o serviço de origem da requisição que chegou a um *Node* específico.
- **Child**: Representa o serviço de destino para o qual a requisição será enviada.
- **CallCount**: Indica o número de requisições e a quantidade de vezes que este Node foi utilizado em outros serviços.
- **ErrorCount**: Informa se ocorreram requisições com erro.

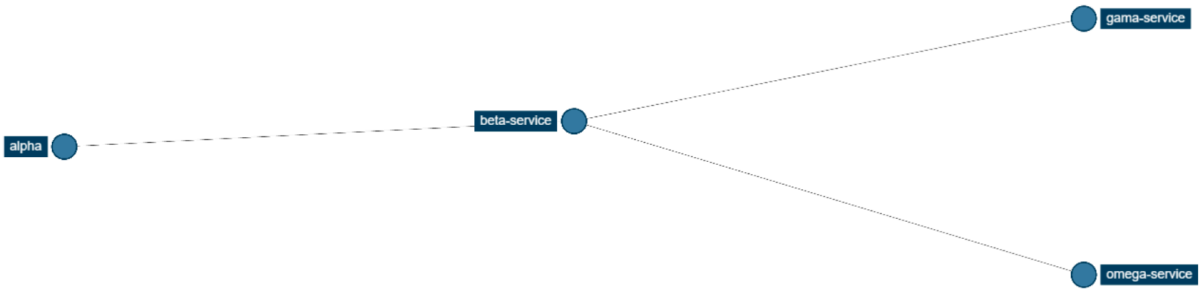
Essa modelagem facilita a análise e a visualização das interações e do desempenho entre os microsserviços na aplicação.

6.6 Implementação do sistema

De acordo com o escopo estabelecido para o projeto foi criado um ambiente de testes com o objetivo de simular a comunicação entre os diferentes módulos,

replicando as condições de um ambiente real de produção, sendo possível visualizar este funcionamento na Figura 18. Esse ambiente serve como base para gerar dados que serão posteriormente analisados pelo módulo responsável pela visualização do sistema. O código-fonte do sistema desenvolvido encontra-se disponível para consulta em um repositório público, hospedado no GitHub, no seguinte endereço: https://github.com/FaelCrios/tcc-projeto.

Figura 18 - Representação visual dos módulos



Fonte: Autoria própria.

Como explicado anteriormente, os módulos foram construídos para simular um ambiente real, com requisições que transitem entre diferentes módulos. Tal funcionalidade pode ser explicada da seguinte forma. O microsserviço *Alpha* faz uma requisição que será encaminhada para o microsserviço *Beta* que posteriormente fará o encaminhamento simultâneo para os microsserviços *Gama* e *Ômega* onde a requisição termina.

Inicialmente, os módulos foram implementados utilizando comunicação via protocolo REST, o que possibilitou a criação de um fluxo de comunicação entre os serviços. Assim, foi possível definir o trajeto que uma requisição deve percorrer entre os módulos, permitindo a análise detalhada do comportamento e da integração dos componentes do sistema.

Portanto, para que o projeto tenha início na máquina instalada, é necessário que sejam executados determinados comandos para que seja acionado o servidor da aplicação, a qual os dados serão gerados e posteriormente deve-se ativar o servidor específico para o projeto Distrace, que irá adquirir os dados necessários e, através da interface gráfica, irá construir as representações visuais..

7. CONCLUSÕES

Neste capítulo, serão tratados os principais diferenciais do software desenvolvido para os outros citados anteriormente, os desafios enfrentados no processo de desenvolvimento do projeto e por fim quais são os pontos pendentes para possíveis melhorias.

7.1 Principais diferenciais

Ao analisar os principais aspectos desenvolvidos neste projeto, é possível identificar que o principal diferencial do software **Distrace**, em comparação aos produtos já existentes no mercado, é sua interface amigável e de fácil compreensão. Esse atributo torna o software acessível até mesmo para usuários que não possuem conhecimentos técnicos na área de tecnologia, permitindo que qualquer pessoa utilize o programa e compreenda o funcionamento do ambiente ao qual ele está integrado.

Outro ponto de destaque é a fácil integração do sistema com plataformas de terceiros para gerar visualizações e análises. Essa funcionalidade amplia significativamente a versatilidade do Distrace, possibilitando que os usuários conectem o software a ferramentas consolidadas para processamento de dados e gráficos. Essa integração permite que as informações coletadas e processadas pelo Distrace sejam apresentadas de forma clara e customizável, atendendo às necessidades específicas de cada usuário.

Com essas características, o **Distrace** se posiciona como uma opção de baixo custo que busca fornecer interpretações claras e de fácil entendimento sobre a estrutura do software ao qual ele está inserido.

7.2 Desafíos enfrentados

Sobre os principais desafios enfrentados no desenvolvimento desta aplicação, é possível citar a dificuldade para escolher uma estrutura de dados que fosse compatível com as necessidades previstas para o projeto. A princípio foi analisado a possibilidade de estruturar os *Nodes* da aplicação em uma estrutura de árvore, porém ao ser colocado em prática verificou-se que era incompatível com as necessidades de expansividade previstas para o projeto, sendo necessário a reorganização da estrutura a cada inserção de um novo microsserviço.

Diante disso, após uma análise aprofundada de outras possibilidades, optou-se pelo uso de uma estrutura de grafo. Essa escolha se mostrou mais adequada, pois facilita a expansão da estrutura para acomodar uma maior quantidade de microsserviços interconectados, eliminando a necessidade de reorganizações frequentes. O grafo também permite representar com maior precisão as relações complexas entre os serviços, garantindo a flexibilidade necessária para o crescimento do sistema.

Contudo, mesmo com a possibilidade de expandir a estrutura utilizada, o sistema ainda possui uma limitação no processo de criação da representação, sendo possível representar um total de 50 microsserviços a partir das requisições obtidas.

Outro desafio enfrentado foi a dificuldade de disponibilizar as rotas utilizadas dentro do ambiente da biblioteca **Zipkin**. Esse problema surgiu da necessidade de acessar as rotas para obter os dados de monitoramento do ambiente do back-end a partir do serviço de coleta oferecido pela biblioteca. No entanto, devido a configurações de segurança, essa ação foi inviabilizada, pois as requisições foram bloqueadas. Portanto, tornou-se possível acessar os dados de monitoramento apenas por meio das telas nativas já disponíveis na biblioteca.

Para solucionar este problema foi necessário fazer a requisição dos dados a partir da tela de requisições, esta que é nativa da aplicação e já possui a liberação nativa do serviço de coleta da biblioteca.

7.3 Possíveis melhorias

Tratando das possíveis melhorias para o projeto, a principal seria transformar o Distrace em um serviço independente da biblioteca Zipkin, permitindo que suas funcionalidades sejam utilizadas de forma autônoma. Para viabilizar essa independência seria necessário desenvolver um serviço de monitoramento exclusivo para o projeto Distrace. No entanto, devido à alta complexidade dessa implementação, essa melhoria foi descartada do escopo deste trabalho.

Outro ponto a ser tratado posteriormente como uma melhoria, é o aumento na capacidade de microsserviços representados de uma só vez. Pensando em um produto que possa ser integrado em qualquer ambiente, tal funcionalidade é imprescindível para o crescimento do projeto.

REFERÊNCIAS

1. AWS. Qual é a diferença entre observabilidade e monitoramento?. Disponível em: https://aws.amazon.com/pt/compare/the-difference-between-monitoring-and-observability/#:~:text=Com%20os%20sistemas%20de%20monitoramento,centenas%20de%20componentes%20do%20servi%C3%A7o. Acesso em: 16 abr. 2024.

2. DIEHL, Stefan. Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Berlin: Springer, 2007.

3. DYNATRACE. What is distributed tracing, and why is it important?. Disponível em: https://www.dynatrace.com/news/blog/what-is-distributed-tracing/. Acesso em: 6 abr. 2024.

4. FORD, Neal; RICHARDS, Mark. Fundamentals of Software Architecture: An Engineering Approach. 1. ed. [S.l.]: O'Reilly Media, 2020.

5. GARLAN, David; SHAW, Mary. An Introduction to Software Architecture. In: VICKERS, Wayne; DEXTER, James (Ed.). Advances in Software Engineering and Knowledge Engineering. 2. ed. Singapore: World Scientific Publishing Company, 1996. p. 1-39.

6. IBM. Benefícios da observabilidade: mais eficiência operacional e inovação. Disponível em: https://www.ibm.com/br-pt/resources/automate/observability-benefits#:~:text=A%20observabilidade%20permite%20que%20eles,e%20resolvam%20quest%C3%B5es%20mais%20rapidamente. Acesso em: 2 dez. 2024.

7. IBM. Qual é o valor da observabilidade para sua organização? Disponível em: https://www.ibm.com/br-pt/resources/automate/observability-benefits. Acesso em: 6 abr. 2024.

8. JAEGER. Jaeger Documentation. Disponível em: https://www.jaegertracing.io/docs/1.56/. Acesso em: 7 abr. 2024.

9. NEWMAN, Sam. Building Microservices: Designing fine-grained systems. 1. ed. [S.l.]: O′Reilly, 0. p. XX-YY.

10. NEW RELIC. New Relic Documentation. Disponível em: https://docs.newrelic.com/. Acesso em: 6 abr. 2024.

11. RICHARDSON, Chris. Microservices Patterns: With examples in Java: 1. ed. [S.l.]: Manning, 2018. p. XX-YY.

12. SOMMERVILLE, Ian. Engenharia de Software. 9. ed. São Paulo: Pearson Prentice Hall, 2011.
13. ZIPKIN. Architecture Open Zipkin. Disponível em: https://zipkin.io/pages/architecture.html. Acesso em: 7 abr. 2024.