Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Observabilidade em Microsserviços com Spring Boot Admin e Stack Prometheus/Grafana

Vitor de Oliveira Araujo Araruna

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Daniel de Paula Porto

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Observabilidade em Microsserviços com Spring Boot Admin e Stack Prometheus/Grafana

Vitor de Oliveira Araujo Araruna

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Daniel de Paula Porto (Orientador)
CIC/UnB

Prof. Dr. Luís Paulo Faina Garcia
CIC/UnB

Prof. Dr. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 1 de janeiro de 2025

Dedicatória

Dedico este trabalho àqueles que, de diferentes formas, me sustentaram e me inspiraram ao longo desta jornada.

Primeiramente, dedico-o aos meus colegas de curso. Compartilhamos a árdua, mas gratificante, experiência de conciliar as exigências da vida profissional com as demandas da vida acadêmica. A conclusão desta etapa, superada por tantos de nós, serve de testemunho à nossa resiliência e disciplina.

Em seguida, dedico esta monografia a todos os amantes da tecnologia, em especial aos simpatizantes da engenharia de software e da gestão de microsserviços. Que este trabalho contribua para o aprimoramento da observabilidade e do monitoramento em arquiteturas distribuídas, um tema que me apaixona profundamente.

Por fim, e com o mais profundo amor, dedico este esforço às duas figuras maternas essenciais em minha vida: minha mãe, Viviane, e minha avó, Sebastianinha. Duas mulheres que, por meio de seus exemplos, me ensinaram os pilares da honestidade, da disciplina e do trabalho. Vocês são minhas referências em terra e minhas maiores inspiradoras.

Agradecimentos

Como cristão, gostaria de agradecer primeiramente a Deus, por ter me capacitado e me dado sabedoria para não desistir do meu trajeto acadêmico, além de ter sido e ainda ser minha fortaleza nos momentos de aflição.

Registro minha gratidão à minha mãe, Viviane, que é minha fortaleza em terra, melhor amiga e minha maior mentora de vida. Sem seu apoio esta jornada acadêmica certamente não teria sido concluída.

Agradeço à minha avó, Sebastianinha, por prestar todo suporte necessário que permitiu conciliar semanas exaustivas de trabalho e estudo. Sem sua ajuda, minha rotina teria sido muito mais difícil e complexa.

Sou grato ao meu pai, Arenilson, e aos meus irmãos, Bernardo e Gabriela, pelo incentivo nos momentos difíceis e pela compreensão diante de minhas ausências em ocasiões festivas, em razão de compromissos profissionais e acadêmicos.

Por fim, agradeço aos professores da Universidade Federal de Santa Catarina e da Universidade de Brasília, cujo ensino e orientação ampliaram meu interesse e apreço pela área da tecnologia, os quais perdurarão ao longo da minha carreira. Agradeço em especial ao meu orientador, Daniel Porto, pela paciência, incentivo e parceria desde as primeiras conversas para definição do tema até a conclusão deste trabalho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho propõe a investigação e o aprimoramento de mecanismos de observabilidade em um ecossistema de microsserviços, utilizando o Spring Boot Admin (SBA) como plataforma central de monitoramento integrado a aplicações de negócio, basicamente simulando dois serviços de um restaurante, contendo produtos e pedidos, em uma arquitetura conteinerizada via Docker Compose. O principal contributo do estudo reside na mitigação de limitações inerentes ao SBA em contextos de produção, especificamente a restrição de visualizações históricas de métricas e a ausência de mecanismos nativos de notificação via WhatsApp. Para tanto, foi implementada uma arquitetura de observabilidade híbrida que integra o SBA com a stack Prometheus-Grafana, viabilizando a análise temporal e quantitativa de métricas técnicas e de Indicadores-Chave de Desempenho (KPIs) de negócio, tais como a análise de padrões de combinação de produtos. A solução proposta foi validada experimentalmente através de demonstrações práticas que comprovaram a eficácia do sistema de alertas multicanal (incluindo notificador customizado para WhatsApp mediante integração com a API Twilio), do gerenciamento operacional de processos assíncronos por meio de acionamento de tarefas agendadas (OrderProcessingTask) e da extensão da interface nativa do SBA com uma Custom View (”Pedidos Atrasados”) para prover visibilidade de métricas de negócio em tempo real. Como trabalhos futuros, são sugeridas a implementação de Rastreamento Distribuído (Distributed Tracing) para diagnóstico de latência end-to-end e o aprimoramento da estratégia de agregação centralizada de logs para garantir a escalabilidade horizontal do sistema de observabilidade.

Palavras-chave: Microsserviços, Observabilidade, Monitoria, Spring Boot Admin, Prometheus, Spring Boot, Grafana

Abstract

This work investigates and enhances observability mechanisms within a microservices ecosystem, employing Spring Boot Admin (SBA) as a central monitoring platform for integrated business applications, specifically simulating two restaurant services related to products and orders in a containerized architecture using Docker Compose. The primary contribution of this study is addressing inherent limitations of SBA in production contexts, particularly the lack of historical metric visualization and the absence of native notification mechanisms via WhatsApp. To achieve this, a hybrid observability architecture was implemented, integrating SBA with the Prometheus-Grafana stack, enabling temporal and quantitative analysis of technical metrics and Key Performance Indicators (KPIs) related to business operations, such as product combination patterns. The proposed solution was experimentally validated through practical demonstrations, which confirmed the effectiveness of a multi-channel alerting system (including a custom WhatsApp notifier using the Twilio API), operational management of asynchronous processes through manual task scheduling (OrderProcessingTask), and the enhancement of SBA's native interface with a Custom View for "Delayed Orders" to provide real-time visibility of business metrics. Future work includes implementing Distributed Tracing for end-to-end latency diagnosis and improving centralized log aggregation strategies to ensure the observability system's horizontal scalability.

Keywords: Microservices, Observability, Monitoring, Spring Boot Admin, Prometheus, Spring Boot, Grafana

Sumário

Lista de Figuras

Capítulo 1

Introdução

A arquitetura de software representa uma organização ideal para um serviço e/ou sistema conduzir suas responsabilidades de uma maneira eficiente. Por muitos anos a arquitetura monolítica foi a mais usada entre as principais empresas de tecnologia do mundo. Ela possui um design de software onde todas as funcionalidades, funções e componentes são integrados em uma única aplicação, ou seja, em uma única base de código. Porém, a crescente demanda por inovação contínua têm levado muitas organizações a adotar arquiteturas de microsserviços, responsável por dividir o sistema em componentes independentes, resultando no abandono dos sistemas monolíticos tradicionais. Como destacado por Johannes Thönes em seu artigo seminal [1], essa transição permite que as empresas construam sistemas mais flexíveis e escaláveis, com cada serviço independente podendo ser desenvolvido, implantado e escalado de forma autônoma. No entanto, essa mudança também introduz complexidades significativas, especialmente em termos de monitoramento e gestão de serviços interconectados.

Essa revolução tecnológica é impulsionada pelo movimento DevOps, que unifica as equipes de desenvolvimento e operações para agilizar a entrega de software, assegurando a preservação da qualidade. Ferramentas de observabilidade e monitoramento são componentes essenciais e praticamente obrigatórios nesse ecossistema, proporcionando a visibilidade necessária para detectar comportamentos inadequados ou imprevistos, além de diagnosticar e resolvê-los rapidamente. O Spring Boot Admin (SBA) [1] é uma ferramenta popular nesse domínio, de fácil configuração e projetada para monitorar serviços baseados em Spring Boot. Ele coleta e exibe métricas essenciais, permitindo que as equipes gerenciem melhor os serviços da sua infraestrutura. Essa coleta de dados é realizada através do "Spring Boot Actuator" [2] , um componente que instrumenta a aplicação para expor um rico conjunto de informações operacionais via endpoints HTTP.

[1]Doc. Oficial Spring Boot Admin: https://docs.spring-boot-admin.com
[2]Doc. Oficial Spring Boot Actuator: https://docs.spring.io/spring-boot/how-to/actuator.html

Apesar de suas capacidades e fácil integração com projetos Spring Boot, o SBA enfrenta limitações que podem restringir sua utilidade em ambientes complexos de microsserviços. Essa complexidade e a natureza distribuída dos sistemas amplificam os modos de falha, tornando a observabilidade uma necessidade que transcende o monitoramento tradicional [2]. O estudo realizado pelo artigo "Monitoring tools for DevOps and microservices: A systematic grey literature review" [3], identifica desafios comuns enfrentados por ferramentas de monitoramento, como a necessidade de personalização e integração com outras plataformas. No caso do SBA, a falta de flexibilidade na personalização de painéis de métricas pode dificultar a análise detalhada, uma vez que seus painéis são extremamente simples e com pouca variedade para realizar buscas e pesquisas customizadas sobre o serviço monitorado diante das métricas coletadas. Além disso, o sistema de alerta padrão possui integração nativa com alguns dos principais canais de comunicação, como o Telegram, Slack, Mail, entre outros. Porém, um grande canal de comunicação, o WhatsApp, não possui uma integração nativa com a ferramenta, não atendendo por completo as necessidades de organizações que requerem notificações em múltiplos desses canais.

1.1 Motivação

A motivação para este trabalho consiste não apenas em explorar o monitoramento de microsserviços em conjunto com a ferramenta SBA, como também proporcionar soluções que potencializam seu uso, além de abordar suas deficiências brevemente mencionadas e como contorná-las. Como resultado, conseguimos adaptar a ferramenta às demandas modernas das organizações contemporâneas. A integração com o Prometheus[3] e Grafana[4], por exemplo, é uma abordagem que oferece dashboards personalizáveis e visualizações mais avançadas, permitindo que as equipes de infraestrutura acessem insights detalhados sobre o desempenho do sistema, além de realizar buscas customizadas sobre o comportamento do sistema observado.

Além disso, a extensão do sistema de alertas para incluir notificações via WhatsApp amplia o alcance das comunicações críticas, assegurando que as equipes estejam sempre informadas, independentemente da plataforma de comunicação utilizada. O uso do WhatsApp, que possui mais de 2 bilhões de usuários ativos, justifica sua adoção para notificações críticas devido à sua onipresença e à alta probabilidade de visualização imediata [4, 5].

Com isso, o estudo que será apresentado neste documento não apenas busca apontar e superar algumas das limitações existentes do Spring Boot Admin, mas também contribui

[3]Site oficial do Prometheus: https://prometheus.io/
[4]Site oficial do Grafana: https://grafana.com/

para o conhecimento a respeito de práticas eficazes de monitoramento em arquiteturas de microsserviços. Conforme apontam Amaro et al. [6], o monitoramento eficaz está diretamente ligado à avaliação do sucesso da adoção do DevOps, indicando áreas que precisam de melhoria. Enquanto o objetivo geral é explorar e diminuir as limitações do SBA, aprimorando suas funcionalidades para suportar ambientes complexos, os objetivos mais específicos se encontram na integração com ferramentas avançadas de monitoramento, na implementação de um sistema de alertas mais robusto e na demonstração de como a *stack* aprimorada permite a análise de Indicadores-Chave de Desempenho (KPIs), transcendendo o técnico, observando também o impacto nos negócios.

1.2 Objetivos

Este trabalho tem como propósito central explorar e aprimorar a observabilidade em um ecossistema de microsserviços, utilizando o Spring Boot Admin (SBA) como ferramenta de análise. A pesquisa visa demonstrar a aplicação prática do SBA, identificar suas limitações em cenários complexos e, por fim, propor e implementar soluções que expandam suas capacidades, alinhando-o às melhores práticas de DevOps.

Para atingir esse propósito, foram estabelecidos os seguintes objetivos para aprimorar a demonstração do estudo:

- **Modelar um ambiente de microsserviços para o estudo**: Para melhor observar e compreender o estudo, foi implementada uma arquitetura de software funcional composta por três aplicações principais para simular um cenário realista de comunicação e dependência:
  - ”Product Service”: responsável pelo cadastro e gerenciamento do catálogo de produtos
  - ”Order Service”: simula a criação de pedidos e depende do ”Product Service” para validar os itens, estabelecendo um cenário de comunicação entre serviços.
  - ”SBA server”: aplicação que hospeda o servidor do Spring Boot Admin. Ela atua como a central de monitoramento, responsável por agregar os dados e fornecer a interface de gerenciamento onde os outros dois serviços são registrados para observação.
- **Superar as limitações de visualização do SBA com painéis personalizados**: Diante da simplicidade dos painéis nativos do SBA, este trabalho objetiva integrar o ambiente com Prometheus, para coleta e armazenamento de métricas, e Grafana, para a criação de dashboards customizados. Isso permite uma análise

visual mais rica e flexível, facilitando a investigação de incidentes e a realização de buscas detalhadas sobre o comportamento dos serviços.

- **Ampliar o sistema de notificações**: Considerando que o sistema de alertas do SBA não possui integração nativa com o WhatsApp, um objetivo prático deste trabalho foi desenvolver uma extensão customizada para o sistema de notificações, implementando um notificador funcional para o WhatsApp.

- **Instrumentar os serviços para a coleta de métricas de negócio**: Para ir além do monitoramento puramente técnico, este trabalho demonstra também a instrumentação de Indicadores-Chave de Desempenho (KPIs). KPIs são métricas quantificáveis que refletem o sucesso de uma organização em atingir seus objetivos estratégicos, traduzindo dados operacionais em visões de negócio. Ferrer et al. [7] define KPIs como um conjunto de parâmetros que permite a avaliação do desempenho do sistema em tempo de execução.

- **Estender a interface do SBA**: Mostrar como a UI do Spring Boot Admin pode ser estendida para exibir informações de negócio. Vamos mostrar como realizamos a injeção de um componente *frontend* na interface do SBA para exibir uma KPI de negócio fundamental: uma lista de pedidos atrasados.

LIMITAÇÃO IDENTIFICADA	SOLUÇÃO IMPLEMENTADA	DETALHES DA IMPLEMENTAÇÃO
Simplicidade e falta de flexibilidade na personalização dos painéis de métricas	Integração com Prometheus e Grafana	Os serviços expõem métricas via Actuator (/actuator/prometheus). O Prometheus coleta essas métricas, e o Grafana (que consulta o Prometheus) cria dashboards customizáveis. O SBA_SERVER utiliza external views para adicionar links diretos na UI do SBA que levam aos dashboards do Grafana.
Ausência de integração nativa com canais de comunicação populares, como o WhatsApp	Desenvolvimento de Notificadores Customizados	Implementação de sistemas de notificação para Discord e WhatsApp (PoC via Twilio), garantindo que os alertas de status sejam projetados para canais externos.

Figura 1.1: Principais limitações

1.3 Organização desse trabalho

Este trabalho está organizado em seis capítulos, dispostos para conduzir o leitor desde a motivação e revisão bibliográfica até a implementação prática, avaliação e proposições futuras.

O Capítulo 2 reúne os trabalhos relacionados, oferecendo um panorama teórico sobre arquiteturas de microsserviços, observabilidade e ferramentas relevantes usadas durante o trabalho (Spring Boot Admin, Prometheus e Grafana), identificando lacunas que motivaram as contribuições deste estudo.

O Capítulo 3 descreve a arquitetura da solução implementada. Apresenta a visão geral e o diagrama de alto nível do ecossistema de contêineres (product_service, order_service, sba_server, Prometheus e Grafana), detalha os componentes (Product Service, Order Service, SBA Server) e explica a orquestração e *scripts* de automação adotados para garantir reprodutibilidade.

O Capítulo 4 constitui o núcleo técnico do trabalho: documenta a implementação das melhorias e integrações. São detalhadas as configurações do ecossistema de monitoramento (Actuator, Micrometer, Prometheus), a criação de dashboards avançados no Grafana, o desenvolvimento de notificadores customizados, a instrumentação de KPIs e métricas de negócio, estratégias de monitoramento de logs, além das funcionalidades do SBA exploradas.

O Capítulo 5 apresenta os resultados e a demonstração prática. Por meio de cenários de teste, medições e capturas de tela, avalia-se o desempenho dos dashboards customizados, a validade do sistema de alertas em tempo real, a análise dos KPIs de negócio, a identificação de padrões de consumo e a operação do painel do SBA.

O Capítulo 6 traz as considerações finais, sintetizando os resultados alcançados, discutindo limitações (por exemplo, ausência de *tracing* distribuído e aspectos de segurança) e propondo trabalhos futuros, tais como integração com OpenTelemetry/Jaeger, adoção de agregação e busca de logs (Loki/ELK) e aprimoramentos na estratégia de coleta e retenção de métricas.

Complementam o trabalho anexos com artefatos como: arquivos de configuração (prometheus.yml, docker-compose.yml, etc), dashboards JSON e trechos relevantes de código (notifier, instrumentação de métricas, etc).

Capítulo 2

Trabalhos Relacionados

O advento da arquitetura de microsserviços revolucionou o desenvolvimento de sistemas ao proporcionar alternativas eficazes para superar os entraves das soluções monolíticas. Desde a metade da década de 2010, estudos como o de Thönes [1] destacaram o potencial dos microsserviços para entregar aplicações compostas por serviços autônomos, cada um com responsabilidade única, o que facilita a escalabilidade, a manutenção e a implantação isolada em ambientes corporativos. Empresas como Netflix e Amazon adotaram essa abordagem para superar as limitações de escalabilidade e entrega contínua enfrentadas por sistemas monolíticos.

Em complemento, o estudo "Microservices: A Systematic Mapping Study", por Claus Pahl e Pooyan Jamshidi [8], realizado na mesma década que o estudo de Thönes mencionado acima, fornece um levantamento abrangente sobre a pesquisa em arquitetura de microsserviços. O artigo é responsável por identificar padrões, desafios e tendências emergentes, consolidando informações fundamentais para pesquisadores e profissionais. Ele apresenta o conceito de microsserviços como um estilo arquitetônico, além de discutir padrões de design como Aggregator, Proxy e Chained Microservice Design Patterns, que são fundamentais para a composição eficaz de serviços.

Estudos sobre a decomposição de aplicações monolíticas em arquiteturas de microsserviços foram surgindo com o passar dos anos, comprovando a tendência e priorização de modelos baseados em microsserviços. Em 2023, uma revisão sistemática foi realizada por um grupo de professores, pesquisadores e aspirantes ao tema. O artigo "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review" [9] aborda metodologias e frameworks, como o "M2MDF", destacando os desafios e as lacunas na decomposição de arquiteturas monolíticas. Os autores identificam a necessidade de ferramentas especializadas para auxiliar essa transição, além de salientar a imaturidade da pesquisa na área.

No âmbito de monitoramento e observabilidade, o estudo "Monitoring tools for DevOps and microservices: A systematic grey literature review" [3] (2024) oferece uma análise abrangente das principais ferramentas e soluções de monitoria utilizadas atualmente. Os autores fizeram o trabalho de analisar 71 ferramentas, destacando funcionalidades essenciais como monitoramento em tempo real, agregação de logs e rastreamento distribuído. A pesquisa enfatiza a importância do monitoramento robusto em ambientes de microsserviços, onde a complexidade dos sistemas distribuídos demanda soluções eficazes para garantir desempenho e confiabilidade. A ferramenta Spring Boot Admin não está entre as 71 ferramentas analisadas no artigo citado, incentivando o seu estudo no presente trabalho, com o objetivo de apresentar algumas de suas limitações e como podemos reduzir esses limites.

No cenário de monitoramento e visualização de dados operacionais, ferramentas como Prometheus e Grafana tornaram-se referência global por sua flexibilidade, escalabilidade e aderência a ambientes de microsserviços. O Prometheus se destaca pela coleta eficiente e armazenamento de métricas temporais, além de seu modelo de consulta poderoso e integração nativa com inúmeros serviços. O Grafana, por sua vez, possibilita a construção de dashboards interativos e customizáveis, permitindo que stakeholders acompanhem, em tempo real, a saúde dos serviços e o cumprimento de metas operacionais e de negócio. A combinação dessas ferramentas é reconhecida em estudos e revisões sistemáticas, como o trabalho do professor Mohammed Daffalla Elradi [10] e Faseeha et al. [11], que definem a observabilidade moderna em torno de três pilares: logs, métricas e traces. O propósito desta tríade é fornecer uma visão mais profunda da causa raiz de gargalos de desempenho e "falhas crosservice" [12]. Ou seja, essa combinação visa potencializar a observabilidade, simplificar o rastreamento de incidentes e ampliar a capacidade de resposta a falhas, preenchendo lacunas deixadas por soluções tradicionais de monitoramento e viabilizando uma abordagem centrada em dados para a gestão de sistemas complexos.

A prática moderna do "Site Reliability Engineering" (SRE) foca em métricas como Latência (p95/p99), Tráfego, Erros e Saturação (Golden Signals), priorizando os indicadores que refletem a experiência do usuário [13, 14], porém, percebe-se que não só apenas métricas de serviço estão em alta hoje em dia. A literatura recente [6] destaca a crescente importância da instrumentação de Indicadores-Chave de Desempenho (KPIs). Além disso, o caráter dinâmico dos sistemas baseados em microsserviços amplificam a necessidade de estratégias robustas de monitoramento, devido aos pontos de falha distribuídos [11]. Para mitigar esses riscos, padrões de resiliência como Circuit Breaker e API Gateway tornam-se essenciais para prevenir falhas em cascata e garantir a estabilidade do sistema [15]. Conforme Amaro et al. [6], os indicadores evoluíram de métricas técnicas tradicionais, como latência ou uso de memória, para indicadores orientados ao negócio, sendo fundamentais

para medir o impacto real das aplicações sobre os objetivos estratégicos da organização. No contexto DevOps, os KPIs são essenciais para avaliar o sucesso da adoção da prática, sendo categorizados em Business, Operational, Change e Cultural [6].

Em arquiteturas distribuídas, a instrumentação correta de KPIs permite que equipes correlacionem diretamente o desempenho técnica com métricas de negócio, viabilizando a tomada de decisão informada e alinhada às prioridades corporativas [6]. No entanto, a implementação desses indicadores é um desafio em microsserviços devido à complexidade de unificar a telemetria em indicadores de negócio inteligíveis [16]. Estudos mostram que o monitoramento eficaz de KPIs, aliado à observabilidade, é determinante para a identificação rápida de gargalos, prevenção de perdas financeiras e validação de hipóteses de negócio em tempo real [7].

Os estudos e ferramentas mencionados acima formam a base para o desenvolvimento deste trabalho, que busca não apenas explorar práticas eficazes de monitoramento, mas também propor melhorias e soluções para as deficiências identificadas na ferramenta analisada. Além disso, o objetivo é modernizar os métodos de monitoria e observabilidade, adotando abordagens alinhadas às utilizadas por grandes empresas de tecnologia, que já incorporam práticas avançadas de observabilidade para obter visão abrangente dos sistemas, resposta rápida a incidentes e maior eficiência operacional em ambientes distribuídos e dinâmicos.

Capítulo 3

Arquitetura da Solução

3.1 Visão Geral da Arquitetura

A finalidade desta seção é descrever a arquitetura global adotada no experimento, cujo objetivo é apresentar a topologia, os fluxos de telemetria e as decisões de projeto empregadas para avaliar e ampliar a observabilidade em um ecossistema de microsserviços.

A solução implementada é composta por três aplicações Spring Boot: ”product_service”, ”order_service” e ”sba_server” — empacotadas como contêineres Docker e orquestradas via Docker Compose [1]. O código-fonte foi organizado em um formato de monorepo[2], uma estratégia que simplifica os pipelines de CI/CD e o acesso a bibliotecas compartilhadas pelos diferentes serviços [17].

```
tcc/
├── docker-compose.yml          # Orquestração de todos os serviços
├── prometheus.yml              # Configuração do Prometheus
├── start_services.sh           # Script de automação
├── grafana/
│   ├── dashboards/
│   │   └── tcc-dashboard.json  # Dashboard customizado
│   └── provisioning/
│       ├── datasources/        # Provisionamento automático do datasource
│       └── dashboards/         # Provisionamento automático de dashboards
├── sba_server/                 # Spring Boot Admin Server
├── order_service/              # Microsserviço de Pedidos
└── product_service/            # Microsserviço de Produtos
```

Figura 3.1: Organização: Diretórios do monorepo

[1]Doc. Oficial Docker Compose: https://docs.docker.com/compose
[2]Diretório raiz do projeto: https://github.com/vitorararuna/tcc

Essa abordagem com orquestração em contêineres, especialmente em ambientes de teste, garante a reprodutibilidade e é uma prática eficiente para o deployment rápido do "monitoring stack" [10]. Para garantir essa reprodutibilidade, o repositório inclui o arquivo docker-compose.yml e o *script* start_services.sh, que automatizam build, criação de imagens e subida dos serviços. Complementam a pilha, ferramentas de observabilidade: Prometheus[3] (configurado via prometheus.yml) para coleta e armazenamento de séries temporais e Grafana [4] para visualização de dashboards provisionados (JSON).

Cada serviço foi instrumentado com Spring Boot Actuator e Micrometer (micrometer-registry-prometheus), expondo, de forma explícita, os endpoints /actuator/health, /actuator/metrics e /actuator/prometheus. O Prometheus está configurado para realizar um scraping periódico desses endpoints (jobs definidos em prometheus.yml) e armazenar as séries resultantes, enquanto o Grafana consome o Prometheus como datasource para dashboards técnicos e de negócio. O sba_server hospeda o Spring Boot Admin (SBA) e funciona como hub operacional central: agrega informações de saúde e configuração das instâncias registradas, exibindo endpoints Actuator de cada cliente e permitindo ações administrativas via UI.

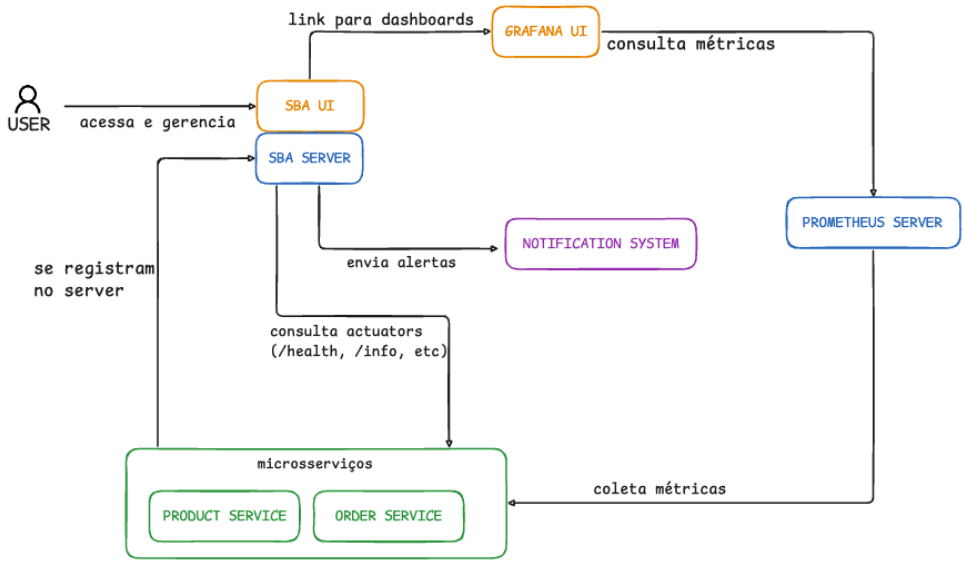

Figura 3.2: Visão de Alto Nível da Arquitetura

A arquitetura foi organizada em dois pilares complementares:

- 1) Pilar de Métricas e Visualização Avançada: instrumentação Actuator/Micrometer → Prometheus (scrape) → Grafana (dashboards técnicos e KPIs de negócio). Neste fluxo foram previstas métricas técnicas (JVM, CPU, memória, http.server.requests)

[3]Doc. Oficial Prometheus: https://prometheus.io/docs/introduction/overview
[4]Documentação Oficial Grafana: https://grafana.com/docs/grafana/latest/datasources/prometheus

e métricas de negócio (ex.: product.create.count; order.product.combinations), todas desenhadas para suportar análises operacionais e comerciais.

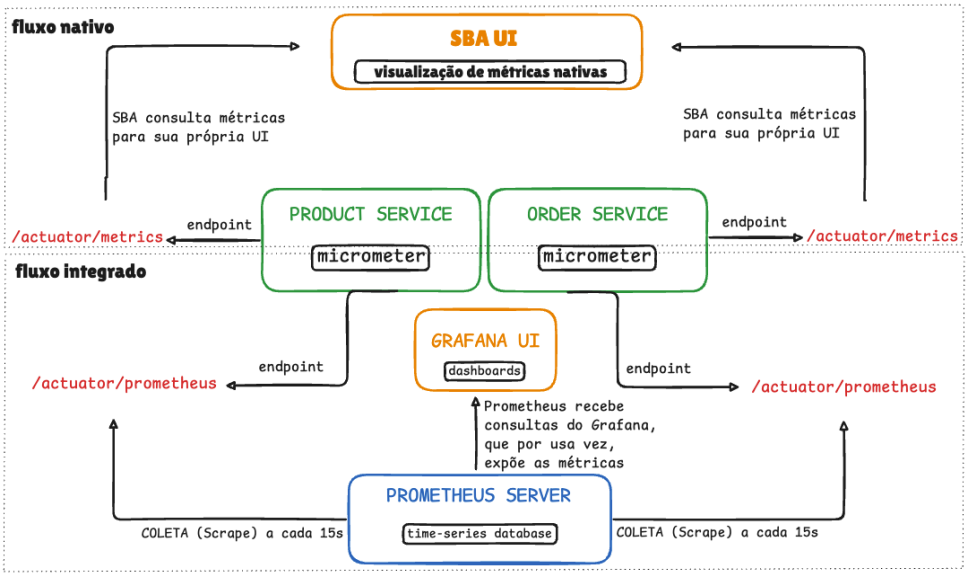

Figura 3.3: Visão do Consumo Periódico das Métricas

- 2) Pilar de Gerenciamento e Notificação: SBA como portal unificado para controle operacional, com notifiers customizados (Discord webhook; PoC de WhatsApp via Twilio) e uso de external views [5](propriedade spring.boot.admin.ui.external-views) para vincular cada instância ao seu dashboard Grafana contextualizado.

3.2 Componentes do Ecossistema

Esta seção apresenta uma análise detalhada dos componentes que constituem o ecossistema de observabilidade implementado, abordando suas responsabilidades funcionais, artefatos técnicos e mecanismos de integração. A arquitetura é composta por três serviços de aplicação (Product Service, Order Service e SBA Server) e uma camada de orquestração e automação baseada em Docker e *scripts* de provisionamento. Cada subseção trata de um componente específico.

- **Product Service**: atua como serviço de catálogo, responsável pela gestão de produtos através de uma API REST que expõe operações CRUD. A persistência é realizada em banco de dados H2 em memória, evitando o anti-padrão de banco de dados compartilhado, pois cada serviço deve idealmente possuir persistência isolada para garantir maior escalabilidade e resiliência [18]. Já a instrumentação de

[5]Extend the UI: https://docs.spring-boot-admin.com/3.5.6/docs/customize/extend_ui

endpoints emprega contadores e timers por endpoint para monitoramento de desempenho. Métricas de negócio customizadas são expostas através dos contadores como ”product.create.count”, ”product.update.count” e ”product.delete.count”, por exemplo, permitindo análise quantitativa das operações realizadas.

- **Order Service**: implementa o fluxo de criação e validação de pedidos, realizando validação de produtos através de comunicação assíncrona com o ”Product Service”, utilizando WebClient. A modelagem de dados contempla as entidades Order e OrderProduct, representando a estrutura hierárquica de pedidos e seus itens, além de uma tarefa (OrderProcessingTask) que é executa periodicamente para identificar e processar pedidos pendentes com mais de três minutos de atraso. Algumas das métricas de negócio implementadas incluem ”order.products.total” para contagem total de produtos e ”order.product.combinations” para análise de padrões de combinação de produtos.

- **SBA Server**: constitui o componente central de monitoramento, implementando o Spring Boot Admin Server para registro e gerenciamento de clientes. O servidor expõe informações dos endpoints Actuator dos serviços registrados e oferece ações de lifecycle para controle operacional. Extensões customizadas foram desenvolvidas para ampliar as capacidades nativas: um notificador Discord (DiscordNotifier) e uma prova de conceito de integração com WhatsApp via Twilio API. A integração com Grafana é realizada através de external views configuradas no SBA, permitindo acesso direto aos dashboards do Grafana a partir da interface do Spring Boot Admin. Endpoints customizados, como o ”/delayed-view”, foram implementados para alimentar views específicas no painel operacional, fornecendo visibilidade em tempo real de métricas de negócio críticas.

- **Orquestração e Automação com Docker e Scripts**: Essa orquestração é gerenciada através de Docker e scripts de provisionamento. O arquivo docker-compose.yml define a topologia de rede, volumes e dependências entre serviços. O script start_services.sh automatiza o processo de build, criação de imagens Docker e inicialização do ecossistema por completo. A configuração do Prometheus (prometheus.yml) define os jobs de scraping e intervalos de coleta de métricas. O provisionamento automático de dashboards no Grafana é realizado através do arquivo ”tcc-dashboard.json”, garantindo reprodutibilidade e versionamento da configuração de visualização.

3.2.1 Product Service: Gerenciamento de Produtos

O product_service constitui um pilar fundamental da arquitetura, encapsulando o domínio de negócio referente ao catálogo de produtos. Sua responsabilidade primária é gerenciar o ciclo de vida da entidade "Product" — incluindo criação, leitura, atualização e exclusão (CRUD) — através de uma API REST. Arquitetonicamente, ele atua como um serviço de dependência *upstream*, sendo consumido pelo "order_service". A persistência dos dados é realizada em um banco de dados em memória (H2), essa foi uma decisão pragmática que simplifica a configuração e a reprodutibilidade do ambiente de experimentação controlado.

Do ponto de vista da observabilidade, o serviço é instrumentado para expor não apenas métricas técnicas padrão (JVM, HTTP), mas também contadores básicos de negócio (product.create.count, por exemplo), que fornecem uma visão quantitativa das operações de escrita. Uma decisão de projeto interessante também foi a inclusão de um endpoint em lote (/batch-details), projetado especificamente para mitigar o risco de consultas N+1 por parte dos consumidores, demonstrando uma preocupação com o desempenho e a eficiência da comunicação entre serviços. Já a configuração de segurança [6] desse serviço, permite acesso irrestrito, adequada ao contexto de desenvolvimento, mas também demonstra a necessidade de mecanismos mais robustos em um cenário de produção, por exemplo.

3.2.2 Order Service: Gerenciamento de Pedidos

O "order_service" representa o coração transacional do ecossistema, sendo responsável por orquestrar a criação e o gerenciamento de pedidos. Como um serviço downstream, ele depende diretamente do "product_service" para validação de itens, estabelecendo um acoplamento síncrono via "WebClient". Esta arquitetura emprega comunicação assíncrona, que é uma prática comum e benéfica em microsserviços [18], uma vez que essa conversação inter-serviços constitui um ponto crítico na arquitetura, cuja latência e disponibilidade impactam diretamente a experiência do usuário, tornando seu monitoramento essencial.

É neste componente que a estratégia de observabilidade atinge seu pleno potencial. Além de ser a fonte das métricas de negócio mais ricas e complexas, este serviço foi projetado para responder a questões comerciais estratégicas. A instrumentação vai além de simples contadores, implementando métricas de alto valor como "order.products.details" (para ranqueamento de popularidade) e, principalmente, "order.product.combinations", uma métrica de análise de cesta de compras para identificar produtos frequentemente consumidos em conjunto. O serviço também hospeda a (OrderProcessingTask), uma tarefa agendada (recurso do SBA na qual entraremos em mais detalhes nos capítulos seguintes)

[6] Doc. Oficial SBA - Foster Security: https://docs.spring-boot-admin.com/3.5.6/docs/server/security

que introduz um padrão de processamento assíncrono à arquitetura, cujo comportamento também é passível de monitoramento.

3.2.3 SBA Server: Central de Monitoramento

O **sba_server** funciona como o hub operacional da arquitetura, materializando o pilar de gerenciamento e notificação. Sua função principal é agregar, de forma centralizada, os dados expostos pelos endpoints Actuator de todos os microsserviços registrados ("product_service" e "order_service"), oferecendo uma interface de usuário unificada para o controle de instâncias. Ele provê uma visão qualitativa e imediata da saúde do ecossistema, permitindo a visualização de configurações, a interação com endpoints e a execução de ações de ciclo de vida, como *shutdown* e *restart*.

A principal contribuição deste componente, no entanto, reside em sua extensibilidade. Ele foi utilizado como plataforma para hospedar notificadores customizados que projetam a observabilidade para canais externos, como o Discord e Whatsapp, transformando eventos de status em alertas acionáveis. Adicionalmente, a arquitetura explora a capacidade do SBA de se integrar a ferramentas externas, como o Grafana, através de "external views", e de consumir endpoints customizados "/delayed-view", demonstrando como a ferramenta pode ser moldada para criar um painel operacional que mescla funcionalidades nativas com contextos de negócio específicos.

3.2.4 Orquestração e Automação com Docker e Scripts

A fundação que sustenta todo o ecossistema é definida pelos artefatos de orquestração e automação.

Os artefatos de orquestração e automação são os responsáveis por sustentar todo o ecossistema. O arquivo docker-compose.yml atua como o manifesto declarativo da arquitetura, definindo os cinco serviços principais (três aplicações, Prometheus e Grafana), suas respectivas imagens de contêiner, configurações de portas e, crucialmente, a rede virtual (monitoring) que permite a comunicação entre eles. É este arquivo que formaliza as conexões entre os componentes, incluindo a configuração que instrui o Prometheus a realizar o *scrape* das métricas expostas pelas aplicações. Esta orquestração, seja com Docker Compose ou ferramentas mais robustas como Kubernetes, é fundamental para o gerenciamento de *deployments*, a garantia de disponibilidade e o balanceamento de carga dos serviços [19].

Complementando a orquestração, o script "start_services.sh" representa uma boa prática de DevOps, funcionando como um artefato de automação que garante a consistência e a reprodutibilidade do ambiente em qualquer máquina. Ele abstrai a complexidade do

ciclo de vida do desenvolvimento, automatizando desde a verificação de dependências e o *build* das aplicações (via Gradle) até a criação das imagens Docker e a inicialização completa do ambiente com um único comando.

Juntos, esses artefatos garantem que qualquer desenvolvedor possa recriar o ecossistema de forma rápida e confiável, permitindo que o foco permaneça na análise e na experimentação das funcionalidades de observabilidade.

Capítulo 4

Implementação, Melhorias e Integrações

O núcleo técnico do trabalho é de suma importância ser apresentado. O objetivo é detalhar a implementação realizada, desde a arquitetura de microsserviços até as soluções propostas para aprimorar a observabilidade e mitigar as limitações identificadas no Spring Boot Admin (SBA). É crucial reconhecer que, no contexto DevOps, artefatos como os Dockerfiles e scripts de infraestrutura são tão críticos quanto o código-fonte da aplicação. Estudos mostram que a qualidade desses artefatos frequentemente apresenta um gap significativo, resultando em vulnerabilidades e falhas de build [20]. Contudo, a solução foi estruturada para ser totalmente reprodutível em qualquer máquina, utilizando Docker e Docker Compose para a orquestração dos serviços e das ferramentas de monitoramento (Prometheus e Grafana).

A documentação a seguir descreve como os componentes foram configurados para formar o ecossistema de monitoramento; como a limitação de visualização do SBA foi superada através da integração com *dashboards* avançados no Grafana; a engenharia por trás dos notificadores customizados (Discord e WhatsApp) que expandem o alcance dos alertas; e, crucialmente, a instrumentação de métricas de negócio (KPIs) que traduzem o desempenho técnico em *insights* comerciais estratégicos. Por fim, será detalhada a exploração das funcionalidades ativas do SBA, como o monitoramento de tarefas agendadas, e como sua interface foi estendida para criar uma experiência de monitoramento unificada.

4.1 Configuração do Ecossistema de Monitoramento (SBA, Actuator, Prometheus)

4.1.1 Instrumentação dos Microsserviços (Clientes SBA):

Os serviços de negócio, **product_service** e **order_service**, foram instrumentados com três dependências críticas que definem seu comportamento no ecossistema de monitoramento:

1. **Registro e Gerenciamento com "spring-boot-admin-starter-client":** Esta dependência tem a função de transformar as aplicações Spring Boot em clientes monitoráveis, habilitando o pilar de Gerenciamento e Notificação.

    - Iniciação do Registro: Ao iniciar, os clientes utilizam esta dependência para se registrarem automaticamente no sba_server, informando sua localização na rede Docker. O próprio sba_server também inclui esta dependência, permitindo o automonitoramento

2. **Exposição de Métricas com "micrometer-registry-prometheus":** Esta dependência garante que as métricas técnicas (como uso de JVM, CPU e latência de HTTP) sejam expostas em um formato compreensível pelo Prometheus, disponibilizando-as no *endpoint* /actuator/prometheus.

3. **Função Agregadora do SBA Server com "spring-boot-admin-starter-server":** A aplicação principal (sba_server) executa o servidor do Spring Boot Admin (SBA) e materializa o pilar de gerenciamento e notificação. Sua "mágica" é habilitada pela dependência spring-boot-admin-starter-server e se manifesta em duas etapas cruciais:

    - Recepção do Registro: O servidor SBA está configurado para receber e listar as instâncias que se anunciam ativamente. Ao receber a requisição de registro dos clientes, ele cria uma entrada para o serviço, sabendo o endereço onde ele está hospedado na rede.
    - Consulta Ativa (Polling): Após o registro, o servidor SBA assume o papel de central de controle. Ele começa a consultar ativamente os *endpoints* Actuator (como /health, /info, e /log) de todos os clientes registrados. É essa consulta periódica que permite ao SBA agregar os dados de saúde e configuração, oferecendo a visão qualitativa e imediata do ecossistema através de sua interface de usuário unificada.

4.1.2 Exposição e Coleta de Dados via Actuator:

Para garantir a total visibilidade, a configuração "management.endpoints.web.exposure.include=*" foi aplicada em todos os serviços. Esta exposição completa dos *endpoints* do Actuator é essencial, pois:

- O SBA utiliza-os para gerenciamento ativo e inspeção centralizada da configuração.
- O Prometheus utiliza especificamente o *endpoint* /actuator/prometheus (de cada um dos serviços) para sua coleta periódica de séries temporais. Essa coleta e armazenamento em sua Time-Series Database (TSDB) serve como a fonte de dados primária para o Grafana, viabilizando os *dashboards* avançados.

4.2 Da Técnica ao Negócio: Instrumentação de KPIs e Métricas Customizadas

O desafio em arquiteturas distribuídas é transformar a telemetria bruta (Bronze) em KPIs de negócio de alto valor (Gold), mantendo a consistência da camada de métricas [16]. Para lidar com os volumes de dados de métricas temporais (como as geradas pelo Micrometer), bancos de dados de série temporal (TSDB), como o Prometheus, são preferíveis a bancos relacionais, pois oferecem desempenho superior em operações de escrita e análise temporal [18]. Embora as métricas técnicas coletadas automaticamente pelo Micrometer (uso de CPU, memória, latência de requisições, etc.) sejam fundamentais para garantir a saúde e a estabilidade da aplicação (observabilidade técnica), elas não respondem a perguntas cruciais sobre o desempenho do negócio. Para preencher essa lacuna, o projeto avançou para a instrumentação de métricas de negócio, ou Key Performance Indicators (KPIs), que traduzem a atividade da aplicação em *insights* estratégicos.

O objetivo desta etapa é ir além de saber se "o serviço está funcionando" e passar a responder perguntas como: "quantos pedidos foram processados na última hora?", "qual o valor total de vendas hoje?" ou "qual o tempo médio para processar um novo pedido?".

4.2.1 Micrometer e o MeterRegistry

A instrumentação de métricas customizadas foi realizada utilizando o "Micrometer" [1], a mesma biblioteca que o Spring Boot Actuator usa internamente. A peça central para essa tarefa é a interface "MeterRegistry" , que é injetada diretamente nos componentes de serviço (como o OrderService) e funciona como uma fábrica para a criação de diferentes tipos de métricas.

[1]Doc. Oficial Micrometer: https://docs.micrometer.io/micrometer/reference/

Os principais tipos de ”medidores” (meters) utilizados foram:

- **Counter:** Utilizado para registrar eventos que só aumentam com o tempo, como o número total de pedidos criados. É ideal para medir volumes e ocorrências.
- **Gauge:** Usado para medir um valor que pode aumentar e diminuir, como o número de pedidos que estão atualmente em processamento.
- **Timer:** Empregado para medir a duração de eventos, como o tempo necessário para processar um pedido do início ao fim.

4.2.2 Exemplo Prático: Instrumentando o order_service

Para materializar o conceito de instrumentação, o order_service foi aprimorado para expor KPIs que medem tanto o desempenho técnica quanto os eventos de negócio. Isso foi feito aplicando-se diferentes tipos de métricas em duas camadas distintas da aplicação: o OrderController e o OrderService.

A peça central para essa tarefa é a interface MeterRegistry do Micrometer, que atua como um registro central e uma fábrica para todos os medidores (métricas) da aplicação. A seguir temos a demonstração de algumas métricas resgatadas no order_service.

Camada 1: Medindo a Latência no Controller com um Timer

A primeira camada de instrumentação é o OrderController, o ponto de entrada para as requisições HTTP. Nesta camada, uma das principais preocupações técnicas é a latência: quanto tempo a aplicação leva para processar uma requisição de criação de pedido do início ao fim?

Para medir isso, um Timer foi utilizado. A implementação foi dividida em dois passos cruciais: a definição da métrica e sua posterior utilização a cada requisição.

Passo 1 - Definição e Registro da Métrica: Para evitar a sobrecarga de recriar a definição da métrica a cada chamada, o Timer é definido uma única vez no construtor do OrderController. Ele é criado usando um ”Timer.builder”, que permite adicionar metadados como uma descrição, e então é registrado com o MeterRegistry. O Timer resultante é armazenado em um campo da classe para uso posterior.

Listagem 4.1: Descritor do construtor do OrderController, onde as métricas são definidas

```
@Autowired
public OrderController(MeterRegistry meterRegistry,
   OrderProcessingTask orderProcessingTask) {
```

```
    this.meterRegistry = meterRegistry;
    this.orderProcessingTask = orderProcessingTask;

    this.getAllOrdersCounter = meterRegistry.counter("endpoint.
       getAllOrders.count");
    this.getAllOrdersTimer = meterRegistry.timer("endpoint.
       getAllOrders.time");

    this.getOrderByIdCounter = meterRegistry.counter("endpoint.
       getOrderById.count");
    this.getOrderByIdTimer = meterRegistry.timer("endpoint.
       getOrderById.time");

    ...
```

Esta abordagem garante que as métricas sejam criadas e registradas apenas uma vez no momento em que o Controller é inicializado, tornando o processo mais eficiente.

Passo 2 - Utilização e Coleta da Métrica no Endpoint: Com o Timer já definido, o método createOrder que lida com as requisições POST pode utilizá-lo para registrar a duração de cada operação. O processo é explícito e claro:

1. O tempo de início (start) é capturado antes de invocar a lógica de negócio.
2. O método ”createOrder(Order order”) é executado.
3. Após a conclusão, a duração total é calculada (System.currentTimeMillis() - start) e registrada no Timer através do método ”.record()”.

Listagem 4.2: Método ”createOrder”

```
@PostMapping
public ResponseEntity<?> createOrder(@RequestBody Order order) {
    createOrderCounter.increment();
    long start = System.currentTimeMillis();

    logger.info("Received␣request␣to␣create␣order:␣{}", order);

    try {
```

```
        Order savedOrder = orderService.save(order);
        logger.info("Order␣created␣successfully␣with␣id:␣{}",
            savedOrder.getId());
        createOrderTimer.record(System.currentTimeMillis() -
            start, TimeUnit.MILLISECONDS);
        return ResponseEntity.ok(savedOrder);
    } catch (RuntimeException e) {
        logger.error("Failed␣to␣create␣order:␣{}", e.getMessage()
            );
        createOrderTimer.record(System.currentTimeMillis() -
            start, TimeUnit.MILLISECONDS);
        return ResponseEntity.badRequest().body(e.getMessage());
    }
}
```

O Timer chamado "**endpoint.createOrder.time**" agora coleta automaticamente a duração de cada chamada a este *endpoint*, além de contar o número total de invocações. Isso permite responder a perguntas como: "Qual a latência média e máxima para criar um pedido?" e "O desempenho está degradando ao longo do tempo?".

Camada 2: Extraindo Inteligência de Negócio no Service

Enquanto o Timer no controller mede o desempenho técnica, a instrumentação na camada OrderService transcende o monitoramento operacional e foca em extrair inteligência de negócio a partir do conteúdo de cada pedido salvo com sucesso.

Em vez de um simples contador de sucesso/falha, a implementação utiliza o MeterRegistry para responder a perguntas de negócio complexas, como "Qual o volume total de vendas?", "Quais são os produtos mais populares?" e "Quais produtos são frequentemente comprados juntos?". Isso é feito através de três métricas customizadas distintas, implementadas dentro do método "save(Order order)".

1. Métrica de Volume Global: order.products.total: A primeira e mais simples métrica é um contador global que mede o volume total de unidades de produtos vendidos.

- Implementação: Um único Counter é definido no construtor da classe "OrderService" e armazenado em um campo (totalProductsCounter). Dentro do método "save(Order order)", após o pedido ser validado e salvo, este contador é incrementado pela quantidade (product.getQuantity()) de cada item no pedido.

Listagem 4.3: Trecho do método **save(Order order)** que incrementa o contador total de produtos

```
for(Order.OrderProduct product : savedOrder.getProducts()) {
  totalProductsCounter.increment(product.getQuantity());

  String productName = productNames.getOrDefault(product.
     getProductCode(), "Desconhecido");

  Counter productCounter = meterRegistry.counter("order.
     products.details",
      "productId", String.valueOf(product.getProductCode()),
      "productName", productName,
      "quantity", String.valueOf(product.getQuantity())
  );
  productCounter.increment(product.getQuantity());
}
```

- Propósito: Esta métrica responde à pergunta de alto nível: ”Qual o volume total de itens vendidos em toda a plataforma?”. Ela serve como um KPI fundamental para medir a atividade geral do negócio em tempo real.

2. Métrica de Popularidade por Produto: order.products.details: Para obter uma visão mais granular, uma segunda métrica é utilizada para analisar a popularidade de cada produto individualmente.

- Implementação: Esta é uma família de contadores dinâmicos. Para cada produto em um pedido, um Counter é acessado (ou criado pelo Micrometer na primeira vez) com o nome ”order.products.details”. Crucialmente, ele é enriquecido com ”tags” que especificam o id, nome e quantidade do produto pedido.

- Propósito: Esta métrica dimensional responde a perguntas mais específicas, como: ”Qual é o produto mais vendido?” e ”Qual a quantidade mais comum para a compra do produto X?”. Ela permite uma análise de padrão de compra por item, vital para o gerenciamento de estoque e para entender o comportamento do consumidor.

3. Métrica de Combinação de Produtos: order.product.combinations: A instrumentação mais inovadora é a que realiza uma análise de cesta de compras em tempo real, buscando identificar quais produtos são comprados em conjunto.

- Implementação: Após salvar o pedido, o serviço itera sobre todos os pares únicos de produtos dentro daquele pedido. Para cada par encontrado, ele incrementa um contador chamado ”order.product.combinations”. Este contador utiliza múltiplas tags para armazenar o nome do par (pair), bem como os IDs individuais de cada produto (product1_id, product2_id).

Listagem 4.4: Trecho do método save(Order order) que define as tags para a metrica

```
for (int i = 0; i < productIds.size(); i++) {
  for (int j = i + 1; j < productIds.size(); j++) {
    Long id1 = productIds.get(i);
    Long id2 = productIds.get(j);

    String name1 = productNames.getOrDefault(id1, "Produto␣
       Desconhecido");
    String name2 = productNames.getOrDefault(id2, "Produto␣
       Desconhecido");

    String pairLabel = String.format("%s␣-␣%s",
        name1, name2);

    meterRegistry.counter("order.product.combinations",
        "pair", pairLabel,
        "product1_id", String.valueOf(id1),
        "product2_id", String.valueOf(id2)
    ).increment();
  }
}
```

- Propósito: Esta métrica é puramente estratégica e responde à pergunta: ”Quais produtos os clientes mais compram juntos?”. A resposta a essa pergunta permite a criação de campanhas de marketing direcionadas, a otimização do layout de produtos em uma loja virtual e a criação de ”combos” ou pacotes promocionais com alta probabilidade de aceitação.

Em conjunto, essas três métricas transformam o sistema de monitoramento de uma ferramenta puramente técnica em uma plataforma robusta de inteligência de negócio, extraindo insights acionáveis diretamente do fluxo de operações da aplicação.

4.2.3 Fechando o Ciclo: Da Métrica ao Dashboard de Negócio

Uma vez que o código é instrumentado, o processo de monitoramento segue o mesmo fluxo já estabelecido:

1. A nova métrica customizada (pedidos.criados) é automaticamente exposta no *endpoint* /actuator/prometheus do order_service.

2. O Prometheus, em seu ciclo de coleta periódica, resgata e armazena essa métrica (*scrapes*) em seu banco de dados de séries temporais (TSDB), sem necessidade de qualquer alteração em sua configuração.

3. Finalmente, no Grafana, é possível criar *dashboards* de negócio que consultam diretamente esses KPIs (mais detalhes sobre esses *dashboards* serão dispostos na próxima seção). Gráficos podem ser construídos para mostrar a evolução do número de pedidos ao longo do dia, a proporção de sucesso vs. falha, e cruzar essa informação com outras métricas técnicas, gerando uma visão verdadeiramente holística do sistema.

Essa transição representa uma evolução fundamental no paradigma de monitoramento. Tradicionalmente, o monitoramento de sistemas opera em um modelo reativo, no qual as equipes agem somente após a notificação de uma falha crítica — como a interrupção de um serviço ou o esgotamento de recursos. O foco, nesse cenário, é a rápida restauração do sistema.

A instrumentação de KPIs, contudo, habilita um modelo proativo. Em vez de apenas alertar sobre falhas já ocorridas, a plataforma passa a fornecer *insights* sobre tendências e anomalias que são precursoras de problemas, como um aumento gradual na taxa de falhas de pedidos. Isso permite que as equipes identifiquem e corrijam a causa raiz antes que ela impacte o usuário final ou escale para uma falha sistêmica.

Dessa forma, a infraestrutura de monitoramento deixa de ser uma ferramenta focada apenas na resposta a incidentes para se tornar um instrumento estratégico para a melhoria contínua e a garantia da qualidade do serviço.

4.3 Superando Limitações Visuais: Dashboards Avançados com Grafana

O Spring Boot Admin (SBA) se destaca ao oferecer uma visão qualitativa e instantânea da saúde do ecossistema, respondendo perguntas como ”qual o status atual do order_service?” ou ”quais as últimas entradas de log?”. Contudo, sua arquitetura é otimizada para a visualização de dados em tempo real, apresentando uma limitação inerente

quando a necessidade é a análise quantitativa e histórica. O SBA não foi projetado para armazenar longos períodos de dados de métricas nem para criar visualizações complexas que correlacionam múltiplas séries temporais.

Para superar essa limitação e habilitar uma análise mais profunda do comportamento do sistema ao longo do tempo, o Grafana foi integrado à pilha de monitoramento. Ele atua como a camada de visualização para os dados coletados pelo Prometheus, transformando as séries temporais em *dashboards* ricos e interativos que permitem a análise de tendências, a identificação de padrões e a investigação de incidentes com base em dados históricos.

4.3.1 Implementação da Integração com o Grafana

A integração do Grafana no projeto foi executada em três etapas principais, garantindo que ele não apenas funcionasse em conjunto com as outras ferramentas, mas que também se tornasse parte de uma experiência de monitoramento unificada.

1. **Orquestração via Docker Compose:** A primeira etapa foi adicionar o Grafana como um serviço no arquivo docker-compose.yml. Esta configuração define como o contêiner do Grafana será executado:

    - Imagem e Rede: Utiliza a imagem oficial "grafana/grafana-oss:latest" e é conectado à mesma rede "monitoring" dos outros serviços, permitindo a comunicação direta com o Prometheus.
    - Persistência de Dados: Um volume (grafana-storage) é montado em /var /lib/-grafana. Isso garante que os *dashboards* e as configurações criadas sejam persistidos mesmo que o contêiner seja reiniciado.
    - Dependência: A cláusula "depends_on: - prometheus" assegura que o Grafana só seja iniciado após o Prometheus estar pronto.

2. **Configuração da Fonte de Dados (Data Source):** Após a inicialização, o passo seguinte é configurar o Grafana para que ele saiba de onde buscar as métricas. Dentro da interface web do Grafana, o Prometheus foi adicionado como uma fonte de dados ("Data Source"). A configuração é direta, bastando apontar a URL para o serviço do Prometheus na rede Docker: http://prometheus:9090. A partir desse momento, o Grafana ganha a capacidade de executar consultas (queries) diretamente na base de dados de séries temporais (TSDB) do Prometheus.

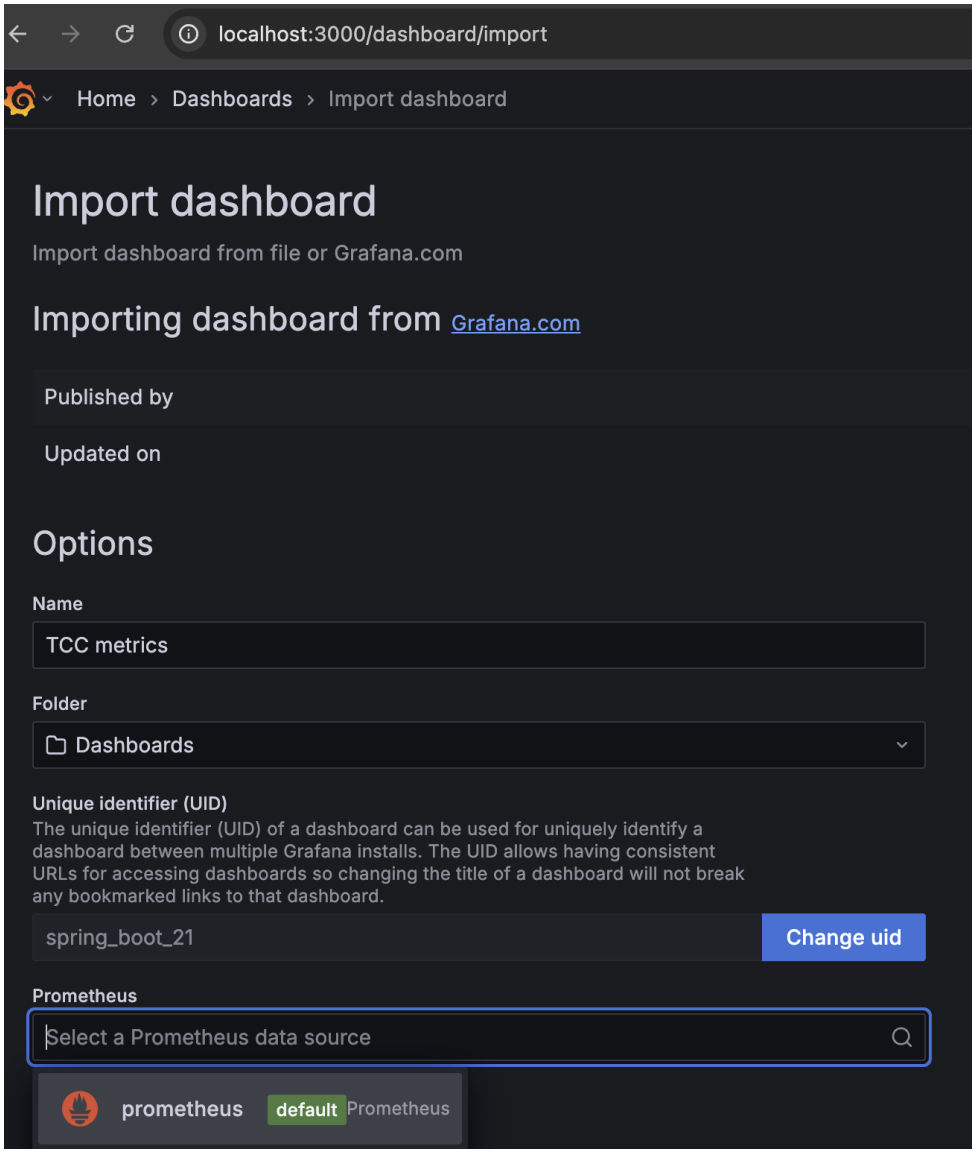

Figura 4.1: Adicionando base do prometheus

3. **Unificação da Experiência na UI do SBA:** O passo final foi projetado para eliminar a fragmentação da experiência do usuário. Para que o desenvolvedor não precisasse memorizar e acessar URLs diferentes, um link direto para os *dashboards* do Grafana foi embutido na interface do Spring Boot Admin. Isso foi alcançado através da propriedade ”spring.boot.admin.ui.external-views” no arquivo application.properties do sba_server.

Listagem 4.5: Trecho onde são adicionadas as configurações das ”external-views”

```
spring.boot.admin.ui.external-views[0].label=\uD83D\uDCCA
   Grafana Dashboards
```

```
spring.boot.admin.ui.external-views[0].children[0].label=\
   uD83D\uDD0D order_service

spring.boot.admin.ui.external-views[0].children[0].url=http:
   //localhost:3000/d/spring_boot_21/dashboard-personalizado-
   tcc?var-application=order_service

spring.boot.admin.ui.external-views[0].children[1].label=\
   uD83D\uDD0D product_service

spring.boot.admin.ui.external-views[0].children[1].url=http:
   //localhost:3000/d/spring_boot_21/dashboard-personalizado-
   tcc?var-

application=product_service

spring.boot.admin.ui.grafana.order=400
```

Com essa configuração, o Grafana deixa de ser uma ferramenta externa e se torna uma extensão natural do Spring Boot Admin, criando um fluxo de trabalho coeso: o SBA é usado para a visão geral e gerenciamento em tempo real, e o Grafana para a análise histórica e detalhada, acessível com um único clique.

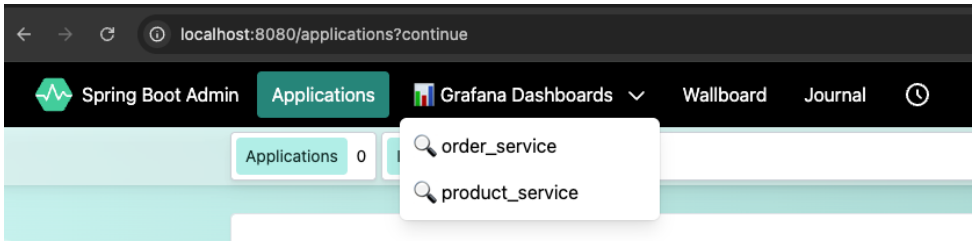

Figura 4.2: Botões com link direto para o Grafana

4.3.2 Provisionamento Automático de Datasource e Dashboard

A configuração manual do Grafana através da interface web apresenta limitações significativas em ambientes que necessitam de reprodutibilidade e versionamento. Quando um contêiner é recriado, todas as configurações realizadas manualmente são perdidas, exigindo reconfiguração repetitiva. Além disso, os identificadores únicos (UIDs) gerados aleatoriamente pelo Grafana para *datasources* impedem a reutilização adequada de *dash-*

boards exportados, uma vez que as referências internas do *dashboard* apontam para UIDs específicos que não existem em novas instalações.

Para resolver essas limitações, foi implementado um sistema de provisionamento automático que configura tanto o *datasources* quanto os *dashboards* durante a inicialização do contêiner Grafana. Esta abordagem garante que o *dashboard* personalizado construído para o projeto esteja disponível imediatamente após a inicialização dos serviços, sem necessidade de intervenção manual.

Configuração do Datasource

O datasource do Prometheus é provisionado automaticamente através do arquivo de configuração "grafana/provisioning/datasources/prometheus.yml", que segue o formato de provisionamento do Grafana (API version 1).

A configuração utiliza um UID fixo "prometheus-ds" em vez de um identificador aleatório gerado pelo sistema. Esta escolha é fundamental para garantir que *dashboards* exportados funcionem corretamente em qualquer ambiente, pois todas as referências internas do *dashboard* apontam para este UID específico. A URL `http://prometheus:9090` utiliza o nome do serviço Docker definido no `docker-compose.yml`, tornando a configuração independente de endereços IP estáticos e aproveitando a resolução de nomes interna da rede Docker. Para garantir consistência adicional na comunicação entre serviços, o Prometheus também possui um endereço IP fixo (`172.19.0.4`) na rede Docker, alinhado com a arquitetura de rede definida para os demais serviços do projeto.

O parâmetro `"isDefault: true"` define este *datasource* como padrão, facilitando a criação de novos painéis. A opção `"editable: true"` permite que administradores modifiquem a configuração através da interface web, se necessário, mantendo flexibilidade operacional. O `jsonData` contém configurações específicas do tipo de *datasource*, neste caso definindo o método HTTP como POST para otimizar consultas ao Prometheus.

Listagem 4.6: Arquivo de provisionamento automático - Prometheus

```
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    uid: prometheus-ds
    access: proxy
    url: http://prometheus:9090
    isDefault: true
```

```
    editable: true
    jsonData:
      httpMethod: POST
```

Configuração do Dashboard

O provisionamento de *dashboards* é configurado através do arquivo "grafana/provisioning/dashboards/dashboards.yml", que instrui o Grafana a importar automaticamente todos os arquivos JSON presentes no diretório `/var/lib/grafana/dashboards`.

Este diretório é montado como volume no `docker-compose.yml`, mapeando o diretório local `./grafana/dashboards` para `/var/lib/grafana/dashboards` dentro do contêiner. Dessa forma, qualquer arquivo JSON de *dashboard* presente no diretório local é automaticamente carregado durante a inicialização do Grafana e organizado na pasta "TCC - Monitoramento", conforme especificado na configuração.

Para garantir compatibilidade total com o sistema de provisionamento, foi necessário atualizar todas as 128 referências ao UID do *datasource* presentes no arquivo `tcc-dashboard.json`. Essas referências foram alteradas de um UID aleatório (gerado durante a criação manual inicial) para o UID fixo `"prometheus-ds"`, garantindo que o *dashboard* funcione corretamente em qualquer ambiente. Além disso, o identificador único do *dashboard* e sua versão foram resetados para permitir uma importação limpa, evitando conflitos com instâncias anteriores.

Listagem 4.7: Arquivo de provisionamento automático - Dashboard

```
apiVersion: 1
providers:
  - name: 'TCC␣Dashboards␣Provisioning'
    orgId: 1
    folder: 'TCC␣-␣Monitoramento'
    type: file
    disableDeletion: false
    editable: true
    options:
      path: /var/lib/grafana/dashboards
```

Benefícios da Abordagem

A implementação do provisionamento automático oferece diversos benefícios significativos para o projeto:

- Reprodutibilidade: Qualquer desenvolvedor pode obter o ambiente completo funcionando automaticamente, sem necessidade de conhecimento específico sobre configuração do Grafana. A simples execução do script de inicialização (`start_services.sh`) resulta em um ambiente totalmente configurado e operacional.
- Versionabilidade: Todas as configurações de *datasource* e *dashboards* são versionadas no sistema de controle de versão (Git), permitindo rastreabilidade completa das alterações e facilitando a colaboração entre membros da equipe.
- Consistência: UIDs fixos eliminam problemas de referência entre *dashboards* e *datasources*, garantindo que *dashboards* exportados funcionem identicamente em diferentes ambientes (desenvolvimento, testes, produção).
- Automação: Zero configuração manual é necessária após a inicialização dos contêineres, reduzindo erros humanos e tempo de setup, além de facilitar a integração em pipelines de CI/CD.
- Infrastructure as Code: O Grafana passa a funcionar seguindo o paradigma de "infraestrutura como código", alinhando-se às melhores práticas de DevOps e permitindo que a infraestrutura de monitoramento seja tratada com o mesmo rigor e versionamento que o código-fonte da aplicação.

Esta abordagem transforma o Grafana de uma ferramenta com configuração manual e propensa a inconsistências em um componente totalmente automatizado e reprodutível, essencial para ambientes de desenvolvimento colaborativo e deploy automatizado.

4.4 Estendendo a UI do SBA: Uma Visão de Negócio para Pedidos Atrasados

Enquanto o Grafana resolve a limitação de análise histórica, a interface nativa do Spring Boot Admin (SBA) ainda se concentra em dados puramente técnicos, carecendo de visibilidade sobre eventos de negócio em tempo real. Para um operador, saber que o order_service está "UP" é importante, mas saber que existem "5 pedidos atrasados precisando de atenção" é uma informação acionável que agrega valor direto ao negócio.

Para preencher essa lacuna, uma ”Custom View” foi desenvolvida, demonstrando a capacidade de extensão do SBA. O objetivo foi criar uma nova página de nível superior (Top-Level View) na interface, dedicada a exibir a lista de pedidos que violaram o Acordo de Nível de Serviço (SLA) de processamento, em outras palavras, pedidos que estão atrasados. Esta abordagem difere das ”External Views” (usadas para os links do Grafana), pois o componente é renderizado nativamente dentro do ambiente do SBA, permitindo uma integração mais profunda.

A implementação foi dividida em três etapas principais: a exposição dos dados no microsserviço, a criação dos componentes de *frontend* e o processo de build que integra as duas partes.

4.4.1 Passo 1: Expondo os Dados de Negócio no order_service

A primeira etapa foi fazer com que o order_service expusesse os dados necessários. A abordagem adotada foi criar um *endpoint* REST padrão dentro do OrderController para expor essa regra de negócio.

Listagem 4.8: Endpoint REST padrão GET /delayed-view

```
@GetMapping("/delayed-view")
  public List<tcc.order_service.dto.DelayedOrderDTO>
     getDelayedOrdersForView() {
    logger.info("Received␣request␣for␣delayed␣orders␣view␣from␣
       Spring␣Boot␣Admin.");

    LocalDateTime cutoffTime = LocalDateTime.now().minusMinutes
       (3);
    List<Order> delayedOrders = orderService.
       findPendingOrdersOlderThan(cutoffTime);

    return delayedOrders.stream()
        .map(this::convertToDto)
        .collect(java.util.stream.Collectors.toList());
  }
```

4.4.2 Passo 2: Implementação do Frontend e Registro da View

O *frontend* da extensão foi implementado como um conjunto de componentes Vue.js, localizados no diretório sba_server/src/main/frontend.

- Componentes Visuais: O arquivo delayed-orders-view.vue contém a lógica e a estrutura visual para renderizar a tabela de pedidos, enquanto o handle-with-label.vue define a aparência do item de navegação no menu principal.
- Registro da View: O arquivo index.js é o ponto de entrada que registra a nova página no "viewRegistry" do SBA. A configuração define esta como uma "Top-Level View", ou seja, um item principal no menu de navegação, e não uma sub-aba de uma instância.

Listagem 4.9: Trecho que utiliza o viewRegistry do SBA

```
SBA.use({
  install({ viewRegistry }) {

    viewRegistry.addView({
      name: 'delayed-orders',
      path: '/pedidos-atrasados',
      component: delayedOrdersView,
      label: 'Pedidos␣Atrasados',
      order: 500,
      handle: handleWithLabel,
    });
  },
});
```

4.4.3 Passo 3: Integração, Build e Fluxo de Dados

A integração final entre o *frontend* customizado e o servidor SBA é orquestrada pelo processo de build e pelo mecanismo de proxy nativo do SBA.

1. **Processo de Build Automatizado:** O arquivo build.gradle do sba_server utiliza o plugin node-gradle para compilar os componentes Vue.js em arquivos JavaScript estáticos. Em seguida, a tarefa processResources presente no arquivo, copia esses

arquivos para o diretório build/classes/java/main/META-INF/spring-boot-admin-server-ui/extensions/custom. Ao iniciar, o sba_server automaticamente descobre e carrega qualquer extensão presente neste caminho.

2. **Fluxo de Dados em Tempo de Execução:** A busca dos dados pelo *frontend* é a parte mais elegante da solução. O componente delayed-orders-view.vue utiliza uma instância ‘axios‘ especial, que é injetada pelo framework do SBA.

Listagem 4.10: Função ’fetchDelayedOrders’ que faz uso do axios para realziar a requisição

```
const fetchDelayedOrders = async () => {
  loading.value = true;
  error.value = null;
  try {
    const response = await axios.get('http://localhost:4040/
       orders/delayed-view');
    orders.value = response.data;
  } catch (e) {
    console.error('[DelayedOrdersView] API call failed!', e);
    error.value = e;
  } finally {
    console.log('[DelayedOrdersView] Fetch process finished.'
       );
    loading.value = false;
  }
};
```

O ‘this.axios‘ não é um cliente HTTP genérico, mas sim uma instância pré-configurada pelo SBA que já possui o ‘baseURL‘ apontando para a URL de serviço da instância selecionada (ex: ‘http://xxx.xx.x.x:4040‘). Portanto, a chamada ”.get(’orders/delayed-view’)” é automaticamente resolvida para a URL completa do *endpoint* no ‘order_service‘, sem que o *frontend* precise conhecer a topologia da rede. O ‘sba_server‘ atua como um proxy transparente para essa comunicação.

Esta implementação demonstra como o Spring Boot Admin pode ser estendido de forma limpa e robusta para ir além de um monitor de saúde técnico, tornando-se um *dashboard* operacional customizado e totalmente integrado, que apresenta dados de negócio relevantes diretamente em sua interface nativa.

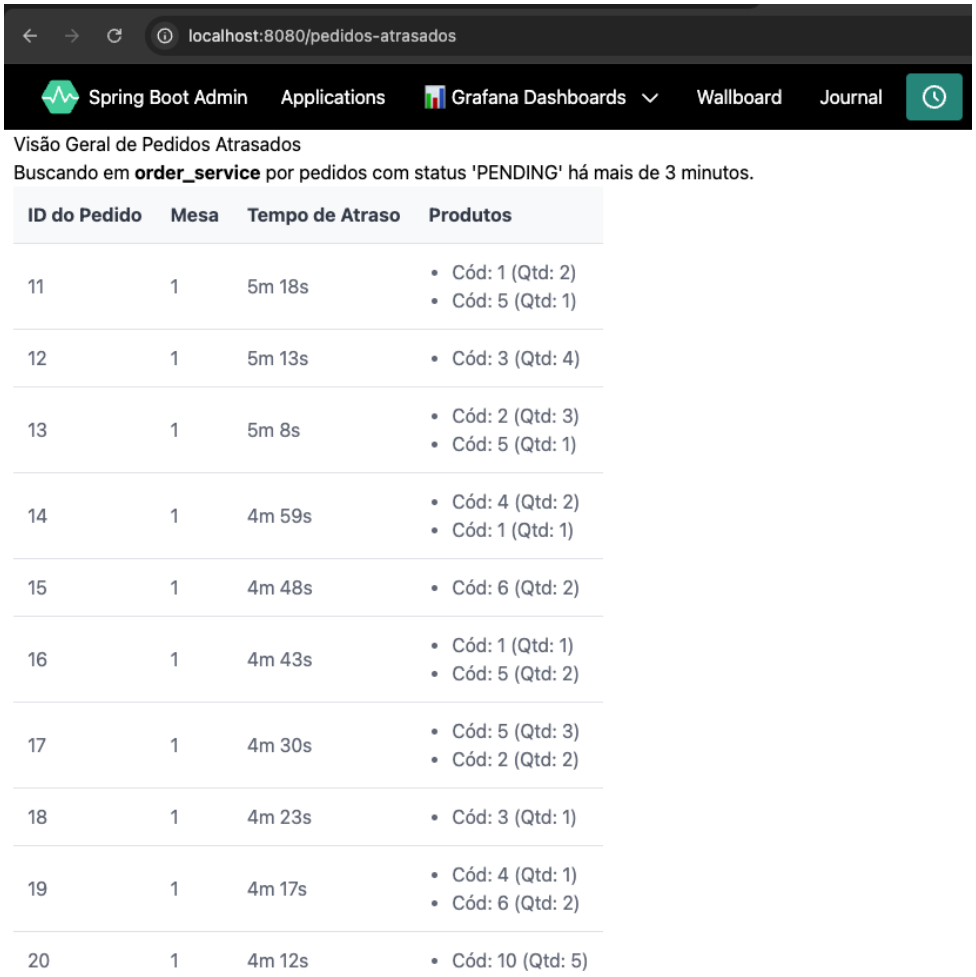

Figura 4.3: Página 'Pedidos Atrasados' na UI nativa do SBA

4.5 Expandindo o Alcance dos Alertas: Notificadores Customizados

A capacidade de notificar as equipes sobre mudanças de status das aplicações (ex: um serviço ficando offline) é uma funcionalidade central do Spring Boot Admin. Por padrão, o SBA oferece suporte nativo a canais como e-mail e algumas plataformas de chat, como Discord, Telegram e Slack por exemplo. No entanto, em um ambiente de desenvolvimento ágil e operações modernas, a eficácia de um alerta está diretamente ligada à sua imediatidade e à sua capacidade de alcançar os desenvolvedores nos canais de comunicação que eles mais utilizam.

Para atender a essa demanda, o subsistema de notificações do projeto foi ampliado para contemplar duas plataformas de comunicação amplamente utilizadas: Discord e WhatsApp. Observa-se que o Discord dispõe de integração nativa, conforme documentado na especificação oficial do Spring Boot Admin, ao passo que o envio de notificações

para WhatsApp constituiu uma extensão implementada neste projeto, visando ampliar os canais de comunicação disponíveis.

4.5.1 Notificações Imediatas via Discord

O Discord evoluiu de uma plataforma para gamers para se tornar uma ferramenta de colaboração popular entre equipes de desenvolvimento. Sua API, baseada em *webhooks*, permite integrações simples e eficientes.

O conceito de webhook e como obtê-lo

Um *webhook* é um mecanismo que permite que sistemas externos enviem mensagens para um canal do Discord de forma automatizada. Na prática, ele funciona como um "endereço de entrega" digital: uma URL única e segura, gerada pelo Discord, que aguarda receber dados (neste caso, uma mensagem de alerta) via uma requisição HTTP POST.

Para obter essa URL, que é a peça-chave da integração, os seguintes passos foram seguidos:

1. **Criação de um Servidor e Canal:** Primeiramente, um servidor foi criado no Discord para o projeto. Dentro deste servidor, um canal de texto específico, nomeado "SBA Notifications", foi criado para centralizar todas as notificações do sistema.

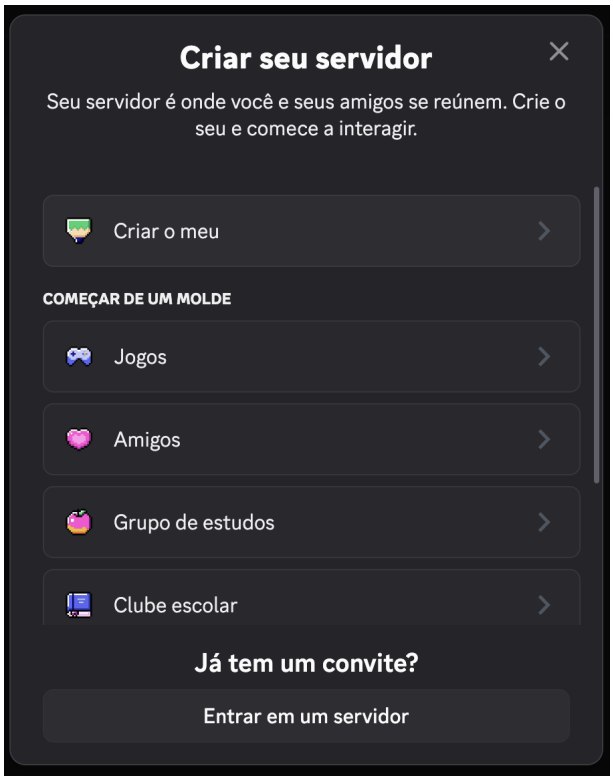

Figura 4.4: Aba para criar servidor no Discord

2. **Acesso às Integrações do Canal:** Nas configurações do canal "SBA Notifications" (acessível pelo ícone de engrenagem), a seção "Integrações" foi selecionada.

Figura 4.5: Aba para adicionar integração ao servidor

3. **Criação do Webhook:** Dentro da seção de integrações, a opção "Novo Webhook" foi utilizada (exemplificada na imagem acima). O Discord permite nomear o *webhook* (ex: "SBA Notifier") e escolher o canal para o qual ele postará as mensagens.

4. **Cópia da URL do Webhook:** Após a criação, o Discord gera e exibe a URL do *webhook*. Esta URL é a informação que deve ser fornecida ao Spring Boot Admin. É importante notar que esta URL é sensível e deve ser tratada como uma credencial, pois qualquer pessoa ou sistema que a possua pode enviar mensagens para o canal.

Configurando e Personalizando o Notificador do SBA

Embora o Spring Boot Admin ofereça uma configuração básica para o notificador do Discord via application.properties, esta abordagem se limita a mensagens de texto simples. Basicamente, a mensagem enviada na configuração básica é um aviso sobre mudança de status dos serviços. Para alcançar um nível superior de detalhe, formatação e inteligência, o projeto implementou um notificador totalmente customizado através da classe DiscordNotification.

Esta abordagem foi escolhida por três motivos principais:

- Riqueza Visual: Permitir o uso de "Embeds" do Discord, que são blocos de mensagens estruturados com cores, títulos, campos e imagens, tornando os alertas muito mais legíveis.

- Lógica Condicional: Habilitar o tratamento de diferentes tipos de eventos. Em vez de apenas notificar sobre mudanças de status, o sistema foi programado para anunciar também quando um novo serviço se registra na plataforma, provendo uma visibilidade completa do ciclo de vida das instâncias.

Figura 4.6: Notificação de aplicação registrada

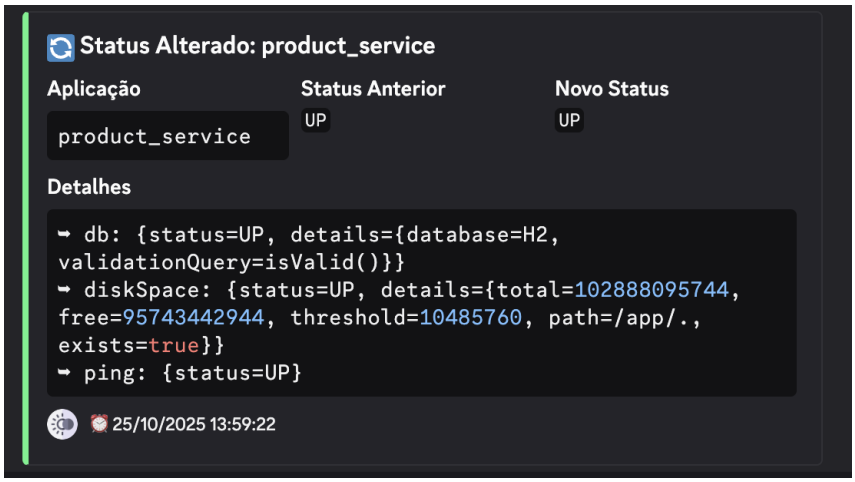

Figura 4.7: Notificação de status de algum serviço alterado

- Controle Total: Ter controle programático sobre a formatação dos dados, o tratamento de erros e a comunicação com a API do Discord.

A implementação se baseia nos seguintes pilares:

1. **Extensão do AbstractEventNotifier:** A classe DiscordNotification é um @Component do Spring que herda de AbstractEventNotifier. Esta é a forma canônica que o Spring Boot Admin provê para a criação de notificadores customizados. A classe sobrescreve o método doNotify, que funciona como um ”dispatcher” central: ele recebe todos os eventos do SBA e os direciona para métodos de tratamento específicos.

2. **Construção de Mensagens Ricas com ”Embeds”:** O ponto central da customização é a construção de um objeto JSON complexo, conhecido como ”Embed”, em vez de uma simples string de texto. A classe possui métodos dedicados para

criar embeds diferentes para cada tipo de evento. O método handleStatusChange, por exemplo, constrói um alerta visualmente rico:

- **Cor Dinâmica:** A cor da borda do alerta muda de acordo com o status da aplicação (verde para *UP*, vermelho para *DOWN*), fornecendo um feedback visual imediato da severidade.
- **Campos Estruturados:** A informação é organizada em campos claros e distintos, como "Aplicação", "Status Anterior" e "Novo Status".
- **Detalhes Formatados:** Os detalhes do erro, que muitas vezes são um bloco de texto JSON, são formatados dentro de um bloco de código (“‘ json... “‘) para facilitar a leitura.
- **Timestamp:** Um rodapé com a data e hora exatas do evento é adicionado, auxiliando na criação de uma linha do tempo durante a análise de um incidente.

4.5.2 Alertas no WhatsApp: Uma Implementação Customizada com a ferramenta Twilio

Enquanto o Discord é excelente para notificações gerais da equipe, o WhatsApp se destaca pela sua onipresença e pela alta probabilidade de visualização imediata, tornando-o o canal ideal para alertas de alta criticidade que exigem atenção urgente.

Diferente do Discord, o Spring Boot Admin não possui suporte nativo para o WhatsApp. Portanto, para viabilizar este canal, foi necessário projetar e desenvolver um notificador customizado. A solução foi construída utilizando a plataforma Twilio [2] como umm *gateway* entre o sba_server e a rede do WhatsApp.

A Plataforma Twilio como Gateway de Comunicação

Jennifer Fei et al. [4] descrevem a Twilio como uma Plataforma de Comunicação como Serviço (CPaaS - Communication Platform as a Service) fundamental para acessar a WhatsApp Business API e gerenciar o fluxo de mensagens para sistemas de pesquisa e notificação. Esta abordagem utiliza o conceito de "Gateway WhatsApp", um sistema projetado para disseminar informações de forma rápida e automática [5]. No âmbito deste projeto, a Twilio abstrai a complexidade do protocolo de comunicação com a rede do WhatsApp, expondo um *endpoint* RESTful que pode ser consumido pela aplicação.

Para utilizar o serviço, foi necessário criar uma conta na plataforma Twilio, obter um número de telefone habilitado para WhatsApp — fornecido diretamente pela Twilio —

[2]Doc. Oficial Twilio - Api: https://www.twilio.com/docs/whatsapp/api

e as credenciais de API (Account SID e Auth Token), fundamentais para a autenticação das requisições.

Especificamente sobre os números envolvidos, a Twilio gera e disponibiliza um número de telefone dedicado para o envio das notificações via WhatsApp, que é adquirido conforme os procedimentos de conformidade regulatória da plataforma. Paralelamente, é necessário cadastrar o número de telefone do destinatário, ou seja, aquele que receberá as notificações, garantindo que este esteja autorizado para tal finalidade.

Em relação aos custos e testes, a Twilio concede um crédito inicial em dólares para uso na fase de desenvolvimento, permitindo o envio de mensagens de teste. Cada notificação enviada pelo WhatsApp é debitada do saldo conforme a política de preços vigente, o que possibilita controlar e estimar os gastos durante o desenvolvimento e operação do sistema.

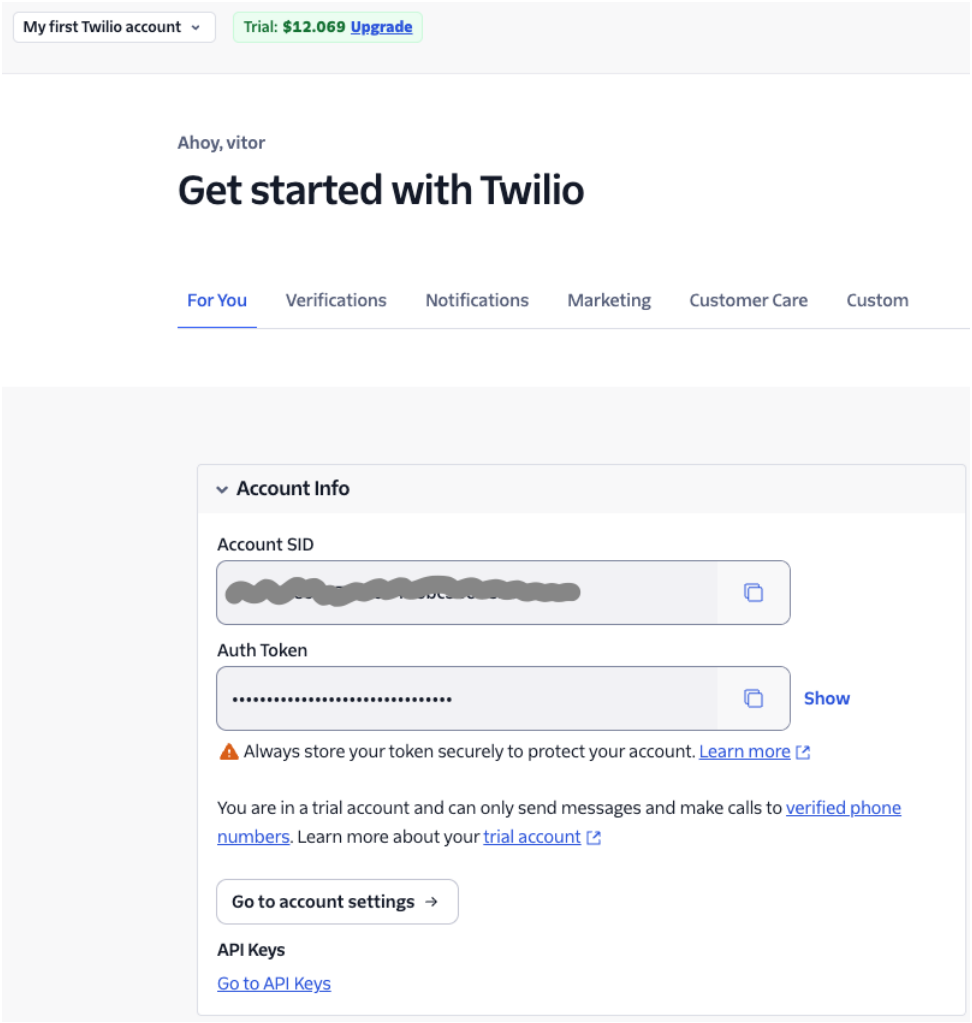

Figura 4.8: Parte do dashboard da Twilio com as informações da conta

Desenvolvimento do Notificador Customizado

A implementação do notificador seguiu um padrão de desenvolvimento similar ao do Discord, mas com a lógica de negócio e comunicação totalmente adaptada para a API do Twilio.

1. **Extensão do AbstractEventNotifier:** Assim como no notificador anterior, a classe WhatsAppNotification herda de AbstractEventNotifier. Seu método doNotify foi configurado para reagir exclusivamente a eventos de mudança de status (InstanceStatusChangedEvent), focando nos alertas mais críticos.

2. **Gerenciamento Seguro de Credenciais:** A segurança é primordial ao lidar com credenciais de API. Todas as informações sensíveis do Twilio (Account SID, Auth Token e números de telefone) foram externalizadas no arquivo application.properties, e são injetadas na classe de forma segura através da anotação '@Value'. Para um ambiente de produção, essas propriedades seriam fornecidas como variáveis de ambiente, evitando que segredos fossem expostos no código-fonte.

```
// Credenciais da conta Twilio (injetadas via variáveis de ambiente):

twilio.account-sid=${TWILIO_ACCOUNT_SID}
twilio.auth-token=${TWILIO_AUTH_TOKEN}

Números de WhatsApp (origem e destino)
twilio.whatsapp.from=${TWILIO_FROM_NUMBER}
twilio.whatsapp.to=${TWILIO_TO_NUMBER}
```

3. **Comunicação com a API do Twilio via 'WebClient':** O núcleo da integração reside no método 'sendToWhatsApp'. Ele utiliza o "WebClient" para construir e enviar uma requisição HTTP POST para a API do Twilio, mas com particularidades importantes:

   - Autenticação: A API do Twilio utiliza "HTTP Basic Authentication". As credenciais (Account SID e Auth Token) são adicionadas ao cabeçalho da requisição.
   - Content-Type: Diferente da API do Discord que usa JSON, a API de mensagens do Twilio espera um corpo de requisição no formato 'application/x-www-form-urlencoded'.

- Corpo da Requisição: O corpo é montado com os campos ‘To‘, ‘From‘ e ‘Body‘, especificando o destinatário, o remetente (ambos com o prefixo ‘whatsapp‘) e o conteúdo da mensagem, respectivamente.

O trecho de código a seguir ilustra a construção dessa requisição complexa:

Listagem 4.11: Trecho do código ’sendToWhatsApp’

```
private Mono<Void> sendToWhatsApp(String message) {
return webClient.post()
    .uri("/Accounts/{accountSid}/Messages.json", this.
       accountSid)
    .headers(headers -> headers.setBasicAuth(this.accountSid,
        this.authToken))
    .contentType(MediaType.APPLICATION_FORM_URLENCODED)
    .body(BodyInserters.fromFormData("To", "whatsapp:" + this
       .toNumber)
        .with("From", "whatsapp:" + this.fromNumber)
        .with("Body", message))
    .retrieve()
    .onStatus(httpStatus -> !httpStatus.is2xxSuccessful(),
       response -> {
      logger.error("␣FALHA␣NO␣ENVIO␣PARA␣O␣WHATSAPP:␣HTTP␣{}"
         , response.statusCode());
      return response.bodyToMono(String.class)
          .flatMap(body -> {
            logger.error("Resposta␣do␣Twilio:␣{}", body);
            return Mono.error(new RuntimeException("Erro␣
               Twilio:␣" + body));
          });
    })
    ...
}
```

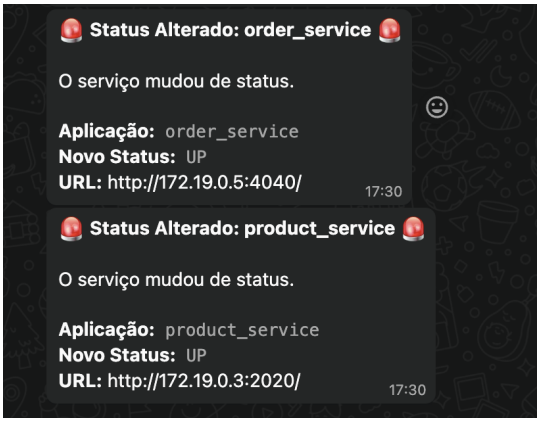

Figura 4.9: Notificações recebidas via WhatsApp

Essa implementação customizada não apenas demonstra a flexibilidade e a capacidade de extensão do Spring Boot Admin, mas também cria um poderoso canal de alerta para incidentes críticos, garantindo que as informações mais importantes cheguem aos responsáveis da maneira mais rápida e eficaz possível, diretamente em seus dispositivos móveis.

4.6 Gerenciamento Ativo e Análise Aprofundada com Recursos Nativos

O Spring Boot Admin (SBA) Server atua como o hub operacional central da arquitetura, materializando o pilar de Gerenciamento e Notificação. Indo além da visualização passiva de status, o SBA fornece ferramentas cruciais para o gerenciamento ativo e a análise aprofundada dos microsserviços, transformando-o de um simples painel de visualização em um console de operações interativo.

Essa capacidade é habilitada pela exposição completa dos *endpoints* do Actuator (configurada via ”management.endpoints.web.exposure.include=*”), permitindo ao SBA interagir diretamente com as aplicações.

4.6.1 Inspeção Centralizada e Gerenciamento do Ciclo de Vida

A principal vantagem do SBA é centralizar a interação com múltiplos *endpoints* do Actuator, provendo um console unificado para operações DevOps.

- Inspeção de Configuração e Ambiente: O SBA oferece acesso em tempo real aos *endpoints* /configprops e /env, permitindo a inspeção detalhada das propriedades de configuração e variáveis de ambiente sem a necessidade de acesso direto ao servidor da aplicação.

- Acesso Centralizado e Download de Logs: O *endpoint* '/logfile' é consultado pelo SBA, permitindo a visualização e o download dos arquivos de log diretamente na interface. Para que esta funcionalidade opere corretamente, a aplicação monitorada precisa ser configurada para escrever em um arquivo (definindo 'logging.file.name') e, para uma melhor legibilidade com cores, a propriedade "spring.output.ansi. enabled=ALWAYS" deve ser ativada.

Figura 4.10: Página de logs com a opção de download

- **Gerenciamento do Ciclo de Vida (Restart e Shutdown):** Uma das capacidades mais poderosas é o gerenciamento remoto. Ao habilitar os *endpoints* '/restart' e '/shutdown', o SBA exibe botões que permitem reiniciar ou desligar um serviço com um único clique. Essa funcionalidade é inestimável para testes de resiliência e para aplicar correções rápidas. É vital que esses *endpoints* sejam protegidos por autenticação e autorização robustas, como o Spring Security, devido ao risco de segurança que representam.

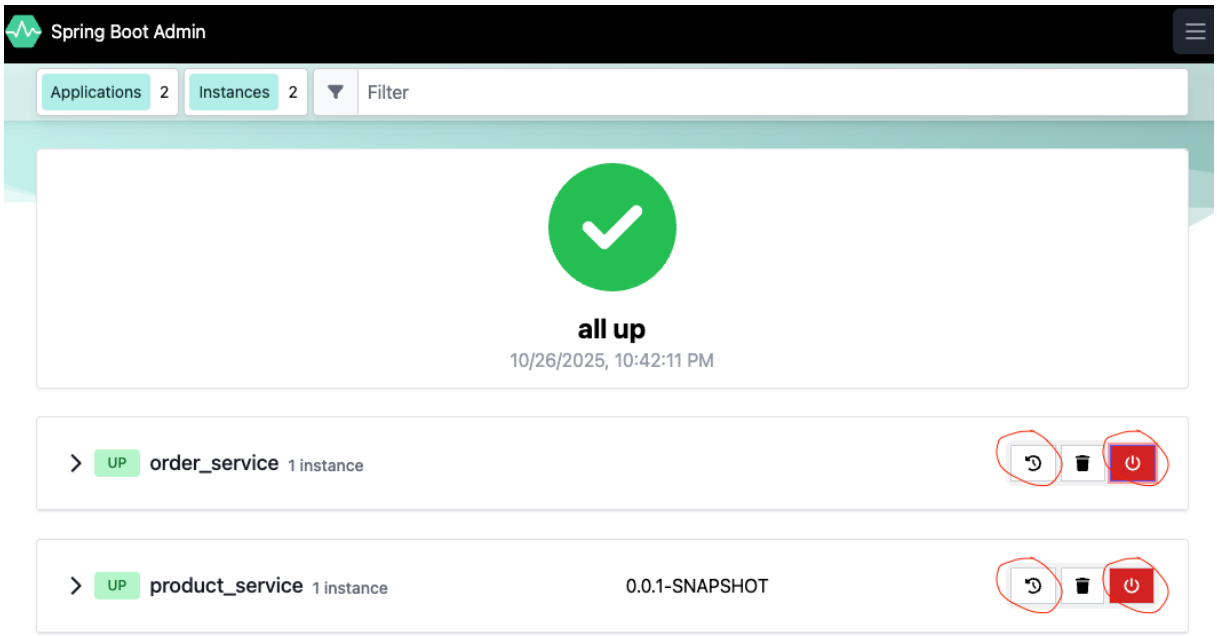

Figura 4.11: Botões restart e shutdown das aplicações

4.6.2 Visualização e Controle de Tarefas Agendadas (@Scheduled)

Aplicações de negócio frequentemente contêm processos que operam de forma assíncrona para monitorar o estado do sistema ou executar lógicas de negócio em segundo plano. Na arquitetura do projeto, essa capacidade é demonstrada no order_service através da classe (OrderProcessingTask).

Esta não é uma tarefa genérica, mas sim um componente que simula o monitoramento de um Acordo de Nível de Serviço (SLA). Sua função é a seguinte:

- A cada 30 segundos, conforme definido pela anotação @Scheduled(fixedRate = 30000), a tarefa é executada.
- Ela consulta o banco de dados em busca de pedidos que estejam em um estado pendente por mais de três minutos.
- Para cada pedido que viola essa condição, a tarefa registra um log de aviso (WARN) de alta visibilidade, alertando sobre o atraso no processamento.

Essencialmente, a (OrderProcessingTask) atua como um fiscal interno, garantindo que os pedidos não fiquem esquecidos no sistema e que os tempos de processamento sejam cumpridos.

O Spring Boot Admin demonstra uma funcionalidade de alto valor operacional ao detectar automaticamente a presença dessa tarefa através do *endpoint* `/actuator/scheduledtasks`. O painel do SBA oferece uma interface dedicada que se torna crucial para o monitoramento deste processo de negócio:

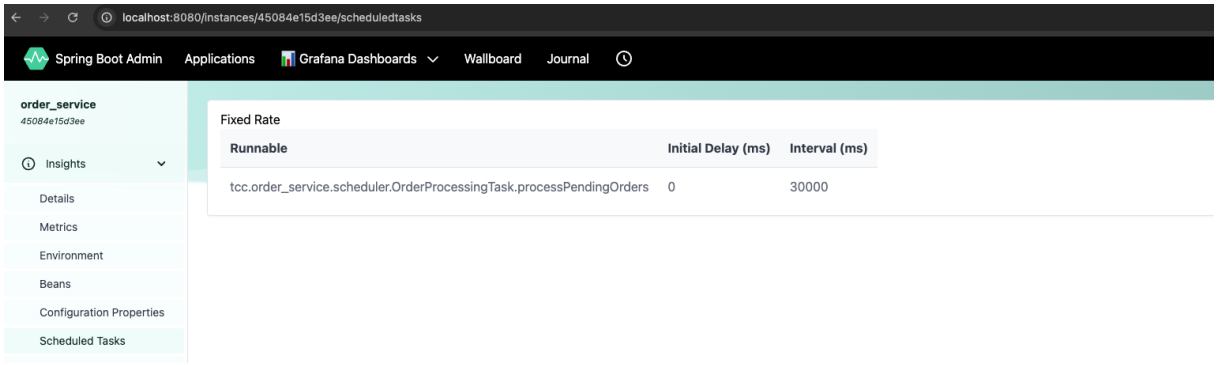

Figura 4.12: Painel "Scheduled Tasks" do SBA

- **Monitoramento de Execução:** O SBA permite que os operadores verifiquem se o "fiscal de SLAs" está operando corretamente, exibindo detalhes como o tipo de agendamento (fixedRate), o intervalo configurado (30000ms) e o método alvo da

tarefa. Se essa tarefa parar de rodar, a visibilidade sobre os atrasos nos pedidos é perdida.

- **Limitação do SBA:** Embora o Spring Boot Admin forneça visualização detalhada das tarefas agendadas, ele não oferece suporte nativo para acionamento manual de tarefas `@Scheduled` através da interface web. Esta é uma limitação conhecida do framework, que foca em monitoramento e visualização, mas não em controle direto de execução.

- **Solução Customizada:** Para superar essa limitação e habilitar acionamento manual sob demanda, foi implementado um *endpoint* REST customizado no `OrderController`: `POST /orders/trigger-scheduled-task`. Este *endpoint* permite que operadores executem a tarefa manualmente a qualquer momento, seja para validação da lógica durante testes ou para verificação imediata de status em operações críticas. A implementação demonstra a flexibilidade de integrar funcionalidades de gerenciamento operacional mesmo quando não suportadas nativamente pelo SBA.

Dessa forma, o SBA fornece a base de monitoramento técnico, enquanto a extensão customizada através de *endpoint* REST oferece o controle operacional necessário sobre este processo de negócio assíncrono e crítico, demonstrando como é possível combinar as capacidades nativas do SBA com soluções customizadas para atender necessidades específicas de gestão operacional.

4.6.3 Análise Quantitativa de Logs via Métricas (Logback)

Embora o acesso ao /logfile seja útil para depuração, o monitoramento eficaz em sistemas distribuídos requer a transformação de eventos de log em métricas de série temporal. Esta abordagem muda o paradigma de monitoramento reativo (ler o log após a falha) para um modelo proativo (detectar tendências anômalas antes que escalem).

A solução foi implementada utilizando a integração nativa do Micrometer com o Logback. Ao configurar um ”appender” especial do Micrometer (io.micrometer.core.instrument.logging.InstrumentedAppender) no arquivo logback-spring.xml, cada evento de log incrementa um contador chamado logback.events.total. A principal vantagem é que este contador possui uma tag level, que permite diferenciar os logs por sua severidade (ERROR, WARN, INFO).

Essa nova métrica é automaticamente exposta no *endpoint* /actuator/prometheus, coletada pelo Prometheus e permite criar visualizações e alertas proativos no Grafana, como um gráfico da taxa de erros por minuto ”rate(logback_events_total{level=”ERROR”}[1m])”, que pode disparar um alerta se exceder um limiar pré-definido.

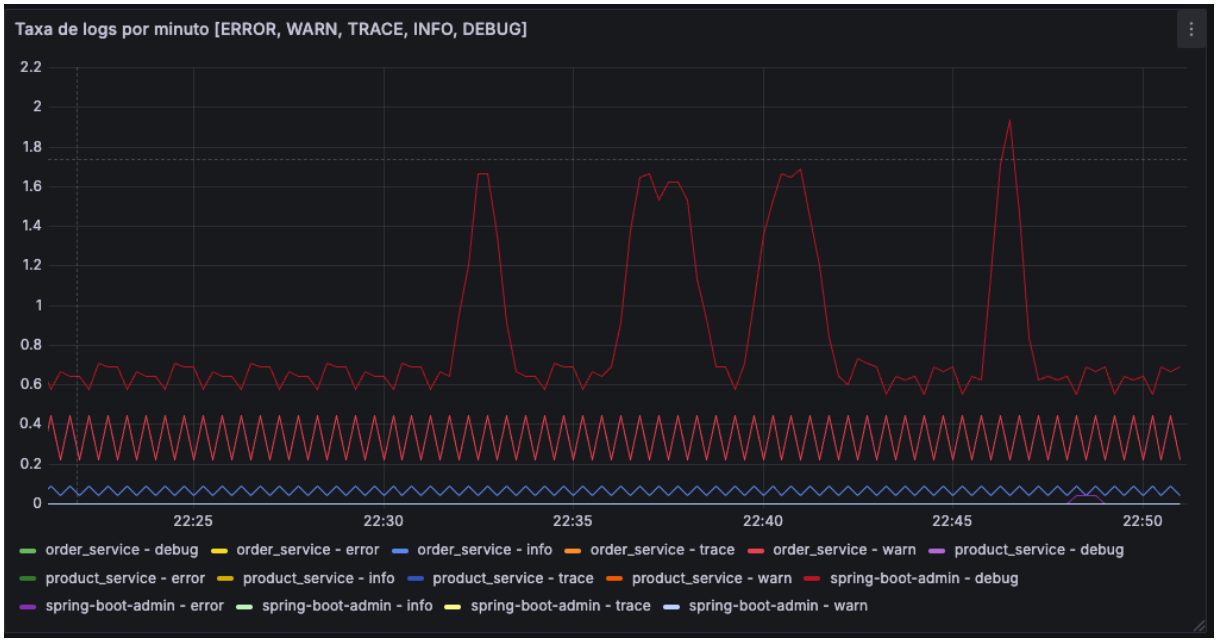

Figura 4.13: Painel de taxa de logs (todos os tipos) por minuto

Capítulo 5

Resultados e Demonstração Prática

Após a detalhada implementação da arquitetura de monitoramento descrita no capítulo anterior, este capítulo se dedica a apresentar e analisar os resultados práticos obtidos. O objetivo é demonstrar como as ferramentas e customizações implementadas se traduzem em uma plataforma de observabilidade robusta, capaz de fornecer desde análises de desempenho técnico até *insights* estratégicos de negócio.

Cada seção a seguir corresponde a uma das funcionalidades implementadas para monitorar e observar o ecossistema de microsserviços envolvendo o sba_server, order_service e product_service. Validando assim, o funcionamento e discutindo o valor agregado por meio de cenários de uso e da análise dos *dashboards* e alertas gerados.

5.1 Análise de Desempenho com Dashboards Customizados no Grafana

O Spring Boot Admin (SBA) é excelente para fornecer uma visão qualitativa e instantânea do estado do sistema, como o status atual ou as últimas entradas de log. No entanto, o SBA apresenta uma limitação inerente para a análise quantitativa e histórica de métricas, pois não é projetado para armazenar longos períodos de dados nem para criar visualizações complexas. A demonstração da integração com o Grafana valida a solução para esta limitação.

- Coleta e Armazenamento: Os microsserviços (order_service e product_service) são instrumentados com Micrometer e Actuator, expondo métricas técnicas (JVM, CPU, latência HTTP) e métricas customizadas no endpoint /actuator/prometheus. O Prometheus realiza o scraping periódico desses dados e os armazena em sua base de dados de séries temporais (TSDB).

Como visto no capítulo anterior, o Grafana já possui uma série de *dashboards* dispníveis no Grafana Lab. Garantindo assim, a visualização de métricas importantes a respeito de serviços. O *dashboard* "Spring Boot 2.1 Statistics" disponibilzado pela ferramenta foi utilizado, onde já conseguimso visualizar informações relevantes sobre os serviços cadastrados, como podemos ver na imagem abaixo:

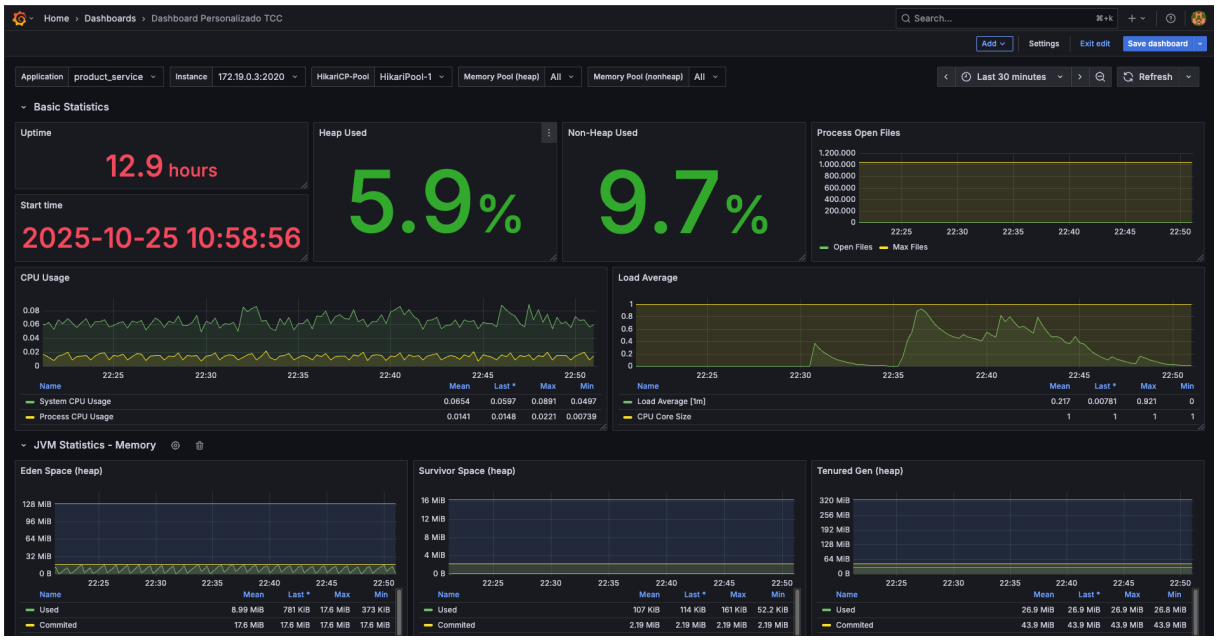

Figura 5.1: Painéis já configurados pelo dashboard disponibilizado pelo Grafana

- Visualização Avançada: O Grafana consome o Prometheus como fonte de dados (http://prometheus:9090), permitindo a criação de *dashboards* customizados para análise de tendências e investigação de incidentes. Para centralizar as visualizações, o *dashboard* disponibizado "Spring Boot 2.1 Statistics" foi utilziado e extendido com as métricas customizadas que foram elaboradas durante o projeto. Dois novos "panels" foram adicionados: "Customizadas - KPIs" e "Customizadas - Performance de endpoints / Logs".

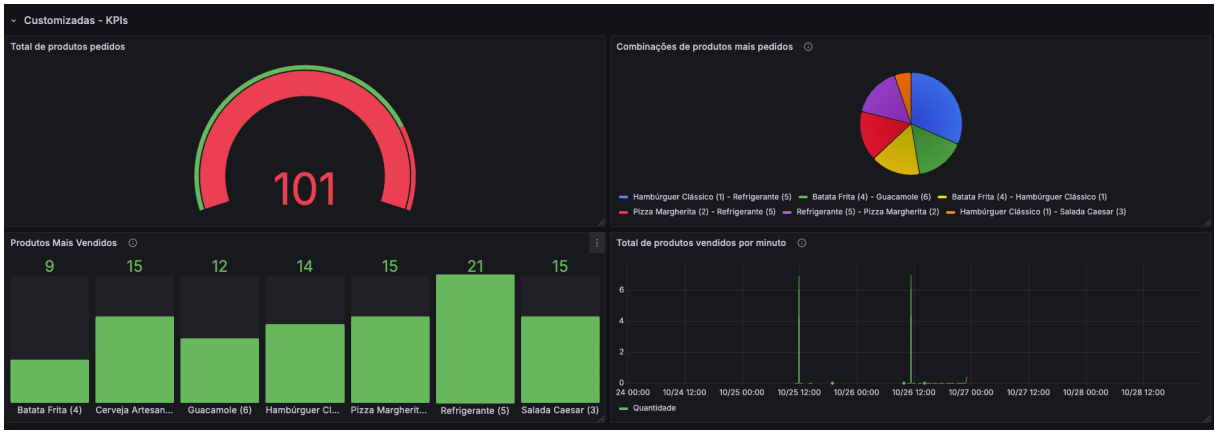

Figura 5.2: Gráficos com os paineis de 'Customizadas - KPIs' - serão mostrados com mais detalhes no decorrer do documento

Figura 5.3: Gráficos com os painéis de "Customizadas - Performance de Endpoints / Logs" - serão mostrados com mais detalhes no decorrer do documento

- Acesso Unificado: Para garantir que o Grafana não seja uma ferramenta fragmentada, a Interface do Usuário (UI) do SBA foi estendida através da propriedade spring.boot.admin.ui.external-views. Essa configuração insere um link direto na barra de navegação do SBA para os *dashboards* específicos do order_service e product_service no Grafana, criando um fluxo de trabalho coeso: o SBA para gerenciamento em tempo real e o Grafana para análise detalhada e histórica.

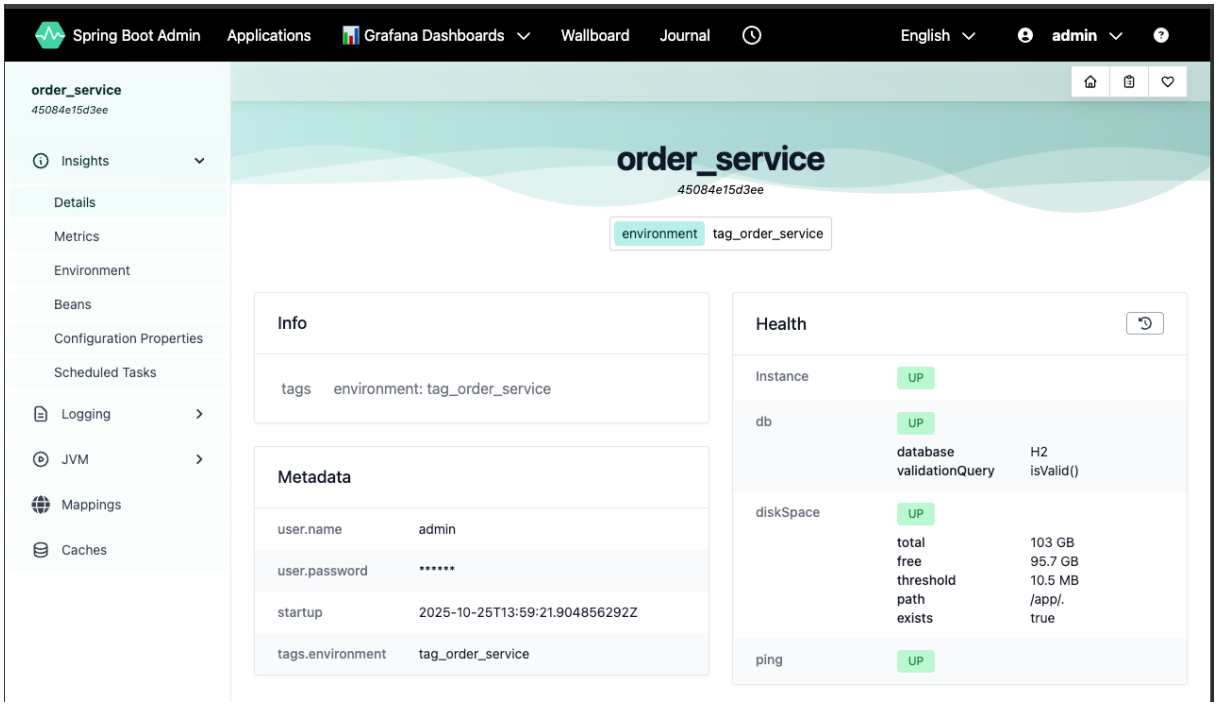

Figura 5.4: Interface nativa do SBA, com uma opção de redirecionamento para o Grafana no menu superior

A visualização no Grafana transforma a análise, permitindo o cruzamento de séries temporais para obter uma visão holística do sistema.

5.2 Validação do Sistema de Alertas em Tempo Real (Discord e Whatsapp)

O projeto expandiu o alcance do sistema de notificações do SBA (originalmente suportando e-mail, Slack, Telegram, etc.) através da implementação de notificadores customizados (Spring Beans que estendem AbstractEventNotifier).

5.2.1 Notificações Imediatas via Discord

A classe DiscordNotification utiliza um *webhook* para enviar alertas:

- Riqueza Visual: O notificador é customizado para usar ”Embeds” do Discord, que são blocos de mensagens estruturados que melhoram a legibilidade.
- Feedback Visual: A cor da borda do alerta é dinâmica, alterando conforme o status da aplicação (por exemplo, verde para UP, vermelho para DOWN, e cinza para OFFLINE), fornecendo um feedback imediato da severidade.

- Eventos Tratados: O notificador trata eventos de mudança de status (InstanceStatusChangedEvent) e também novos registros de aplicação (InstanceRegisteredEvent), provendo visibilidade sobre o ciclo de vida das instâncias.

5.2.2 Alertas de Alta Criticidade via WhatsApp (Twilio)

Para superar a ausência de suporte nativo ao WhatsApp, o projeto implementou o WhatsAppNotification.java utilizando a plataforma Twilio como gateway de comunicação.

- Mecanismo: O notificador reage exclusivamente a eventos de mudança de status, enviando uma requisição HTTP POST para a API da Twilio.
- Protocolo: A comunicação exige HTTP Basic Authentication com credenciais (Account SID e Auth Token) e utiliza o tipo de conteúdo application/x-www-form-urlencoded no corpo da requisição, adaptando-se às exigências da Twilio, que usa os campos To, From, e Body.
- Segurança: As credenciais e números de WhatsApp são externalizados no application.properties e injetados de forma segura via @Value.

Essa implementação customizada expande o alcance dos alertas para um canal de comunicação de alta probabilidade de visualização imediata.

5.3 Análise Prática dos KPIs de Negócio

A verdadeira força da arquitetura de observabilidade implementada reside na capacidade de traduzir a atividade da aplicação em Indicadores-Chave de Desempenho (KPIs) de negócio, visualizados em um *dashboard* customizado no Grafana. Enquanto as métricas técnicas respondem ”como” o sistema está operando, os KPIs de negócio respondem ”o quê” o sistema está produzindo de valor.

O order_service foi o foco dessa instrumentação, e os painéis a seguir demonstram como as métricas customizadas geram *insights* estratégicos a partir dos pedidos processados.

5.3.1 Visão Macro: Volume Total e Taxa de Vendas

Os primeiros painéis do *dashboard* de negócio oferecem uma visão macro da atividade comercial da plataforma, utilizando a métrica **order_products_total**.

- **Painel 1: Total de Produtos Pedidos**

- Visualização: Um painel do tipo ”Gauge” (Medidor).
- Query PromQL: order_products_total
- Análise de Resultado: Este painel exibe um número único e de alto impacto: o total acumulado de unidades de produtos vendidos desde que a aplicação foi iniciada. Ele serve como o principal indicador de volume de negócio, fornecendo uma medida imediata e quantificável de toda a atividade comercial.

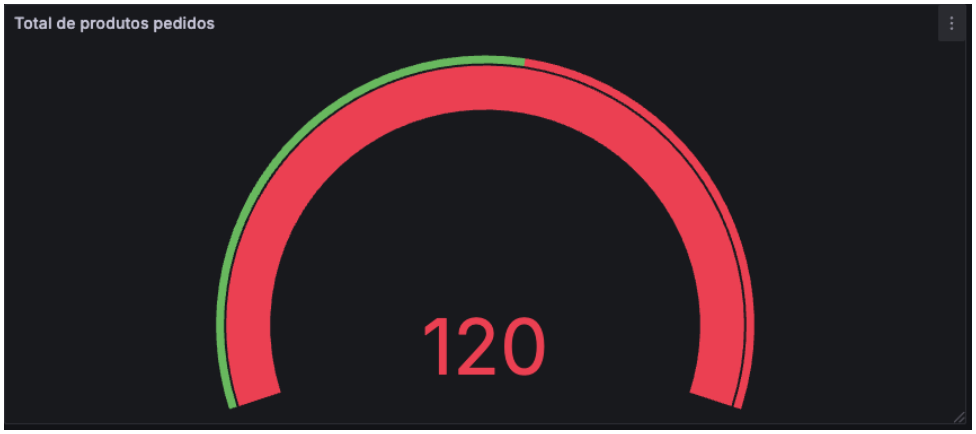

Figura 5.5: KPI de total de produtos pedidos

- **Painel 2: Total de Produtos Vendidos por Minuto**
  - Visualização: Um gráfico de ”Time series” (Série Temporal).
  - Query PromQL: rate(order_products_total[5m]) * 60
  - Análise de Resultado: Este gráfico mostra a taxa de vendas em tempo real. A função rate() do Prometheus calcula a taxa de crescimento do contador por segundo, e a multiplicação por 60 a converte para uma unidade de negócio mais intuitiva (itens por minuto). Este painel é crucial para identificar horários de pico, entender a sazonalidade das vendas e avaliar o impacto imediato de uma campanha de marketing.

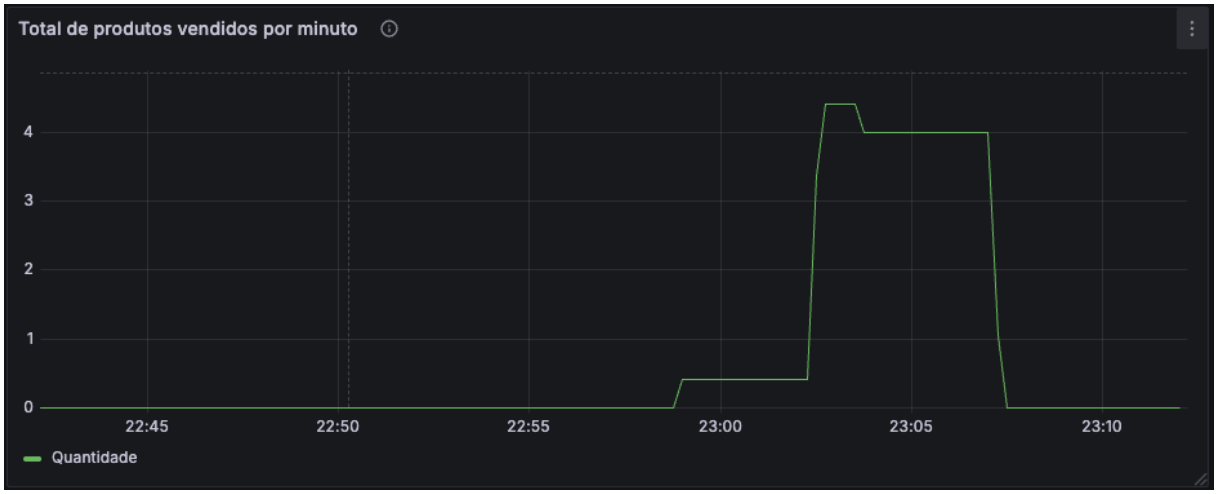

Figura 5.6: KPI de total de produtos vendidos por minuto

5.3.2 Análise de Padrões de Consumo

Os painéis seguintes utilizam as métricas dimensionais para extrair inteligência de negócio mais profunda, focando nos padrões de compra dos clientes.

- **Painel 3: Produtos Mais Vendidos**
  - Visualização: Um gráfico de ”Bar Gauge” (Gráfico de Barras).
  - Query PromQL: topk(10, sum(order_products_details_total) by (productName))
  - Análise de Resultado: Este painel responde a uma das perguntas mais importantes para o negócio: ”Quais são os nossos produtos mais populares?”. A query utiliza sum(...) by (productName) para agregar todas as vendas por nome de produto e topk(10, ...) para filtrar e exibir apenas os 10 mais vendidos. Esta visualização é fundamental para decisões de gerenciamento de estoque, destaque de produtos na interface e planejamento de produção.

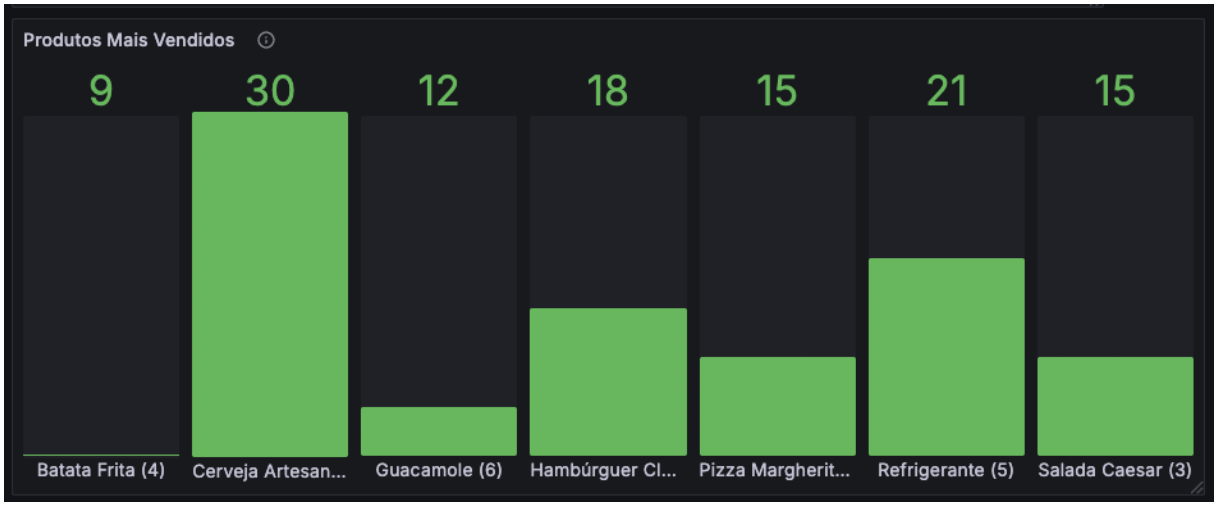

Figura 5.7: KPI de quais são os produtos mais vendidos

- **Painel 4: Combinações de Produtos Mais Pedidos**
  - Visualização: Um painel do tipo ”Pie Chart” (Gráfico de pizza).
  - Query PromQL: topk(10, sum(order_product_combinations_total) by (pair))
  - Análise de Resultado: Este é o painel mais estratégico, realizando uma análise de cesta de compras em tempo real. Ele exibe os pares de produtos que são mais frequentemente comprados juntos. A visualização em tabela é ideal para mostrar os nomes completos dos pares (ex: ”Pizza - Refrigerante”). Este insight é extremamente valioso para a criação de ”combos” promocionais, para o sistema de recomendação (”clientes que compraram X também compraram Y”) e para otimizar o layout de produtos em uma loja virtual.

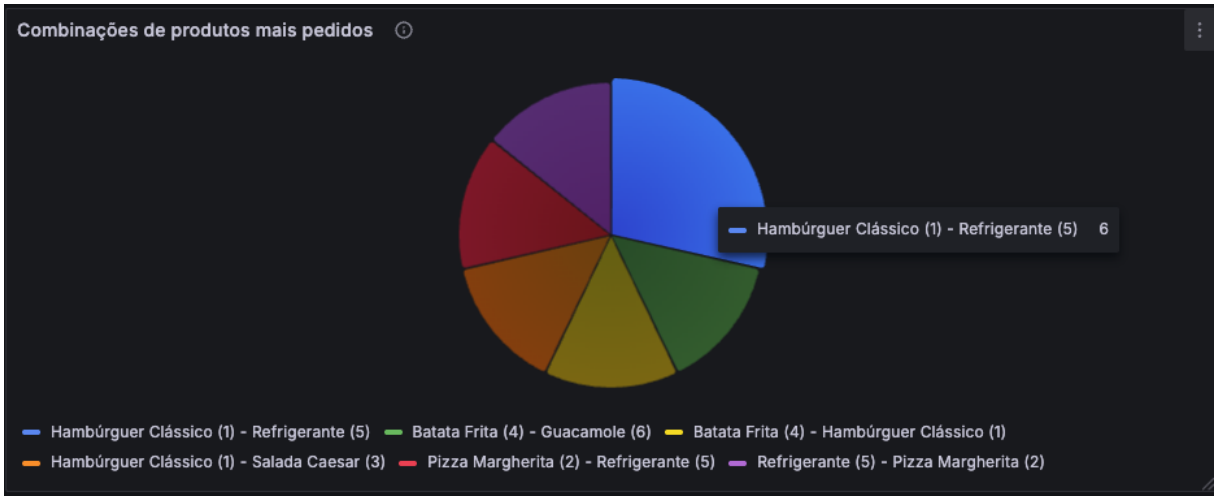

Figura 5.8: KPI de combinação dos produtos mais vendidos

Em conjunto, estes quatro painéis demonstram a transição bem-sucedida de um monitoramento puramente técnico para uma plataforma de inteligência de negócio em tempo real, utilizando a instrumentação customizada para extrair valor estratégico diretamente das operações do dia a dia.

5.4 Demonstração do Gerenciamento de Tarefas & Extensão Visual no Painel Nativo

O Spring Boot Admin (SBA) demonstra seu valor operacional ao oferecer controle direto sobre os processos de negócio assíncronos e ao ser extensível com visões de negócio customizadas. Esta seção valida duas funcionalidades cruciais integradas ao painel nativo do SBA: o gerenciamento de tarefas agendadas (@Scheduled) e a injeção de uma Custom View para a análise de Pedidos Atrasados.

5.4.1 Gerenciamento de Tarefas Agendadas (@Scheduled)

O SBA demonstrou capacidade de detectar e exibir tarefas agendadas através do *endpoint* `/actuator/scheduledtasks`, como evidenciado pela tarefa `OrderProcessingTask` no order_service. A interface do SBA apresenta informações detalhadas sobre o agendamento (tipo, intervalo, método alvo), permitindo monitoramento em tempo real do "fiscal de SLAs" que verifica pedidos pendentes há mais de três minutos.

Embora o SBA não ofereça acionamento manual nativo de tarefas `@Scheduled`, foi implementado um *endpoint* REST customizado (`POST /orders/trigger-scheduled-task`) que permite execução sob demanda. Esta solução demonstra a flexibilidade de estender as capacidades do SBA com funcionalidades customizadas quando necessário para operações críticas.

A execução da tarefa, seja automática (a cada 30 segundos) ou manual (via *endpoint*), gera logs estruturados que facilitam o rastreamento de pedidos que violam o SLA, como ilustrado abaixo.

```
(22025-12-02 02:04:22.207  INFO 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        : Total de pedidos atrasados: 5
(22025-12-02 02:04:52.194  INFO 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        : Verificando pedidos pendentes com mais de 3 minutos...
(22025-12-02 02:04:52.207  WARN 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        :  ATENÇÃO - PEDIDO ATRASADO !!!  Pedido ID 1 está atrasado (criado em: 2025-12-
02T01:48:37.961918).
(22025-12-02 02:04:52.207  WARN 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        :  ATENÇÃO - PEDIDO ATRASADO !!!  Pedido ID 2 está atrasado (criado em: 2025-12-
02T01:48:38.217381).
(22025-12-02 02:04:52.209  WARN 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        :  ATENÇÃO - PEDIDO ATRASADO !!!  Pedido ID 5 está atrasado (criado em: 2025-12-
02T01:48:38.356568).
(22025-12-02 02:04:52.209  WARN 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        :  ATENÇÃO - PEDIDO ATRASADO !!!  Pedido ID 7 está atrasado (criado em: 2025-12-
02T01:48:38.474722).
(22025-12-02 02:04:52.209  WARN 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        :  ATENÇÃO - PEDIDO ATRASADO !!!  Pedido ID 10 está atrasado (criado em: 2025-12-
02T01:48:38.571077).
(22025-12-02 02:04:52.209  INFO 1 --- [   scheduling-1] t.o.scheduler.OrderProcessingTask        : Total de pedidos atrasados: 5
```

Figura 5.9: Logs disparados pela task

5.4.2 Extensão Visual: A Visão de Negócio "Pedidos Atrasados"

O projeto superou a limitação do SBA em exibir dados de negócio em tempo real através da implementação de uma Custom View. O resultado prático é a injeção da nova página "Pedidos Atrasados" como um item principal (Top-Level View) no menu de navegação do SBA.

A demonstração prática dessa extensão valida o fluxo de dados customizado:

1. **Exposição dos Dados:** O order_service expõe os dados de negócio (pedidos pendentes há mais de 3 minutos) através de um *endpoint* REST padrão em seu OrderController: GET /orders/delayed-view.

2. **Proxy Transparente:** O componente *frontend* (delayed-orders-view.vue) utiliza uma instância axios especial, injetada pelo framework do SBA. Essa instância de cliente HTTP é pré-configurada com a baseURL da instância do serviço selecionado, fazendo com que a chamada axios.get('orders/delayed-view') seja resolvida para a URL correta do order_service.

3. **Unificação:** O sba_server atua como um proxy transparente para essa comunicação, permitindo que a Custom View exiba uma lista acionável de pedidos que violaram o SLA diretamente na interface nativa, mesclando a saúde técnica com o contexto de negócio.

Em resumo, a demonstração prova que o SBA pode ser transformado em um *dashboard* operacional customizado e totalmente integrado.

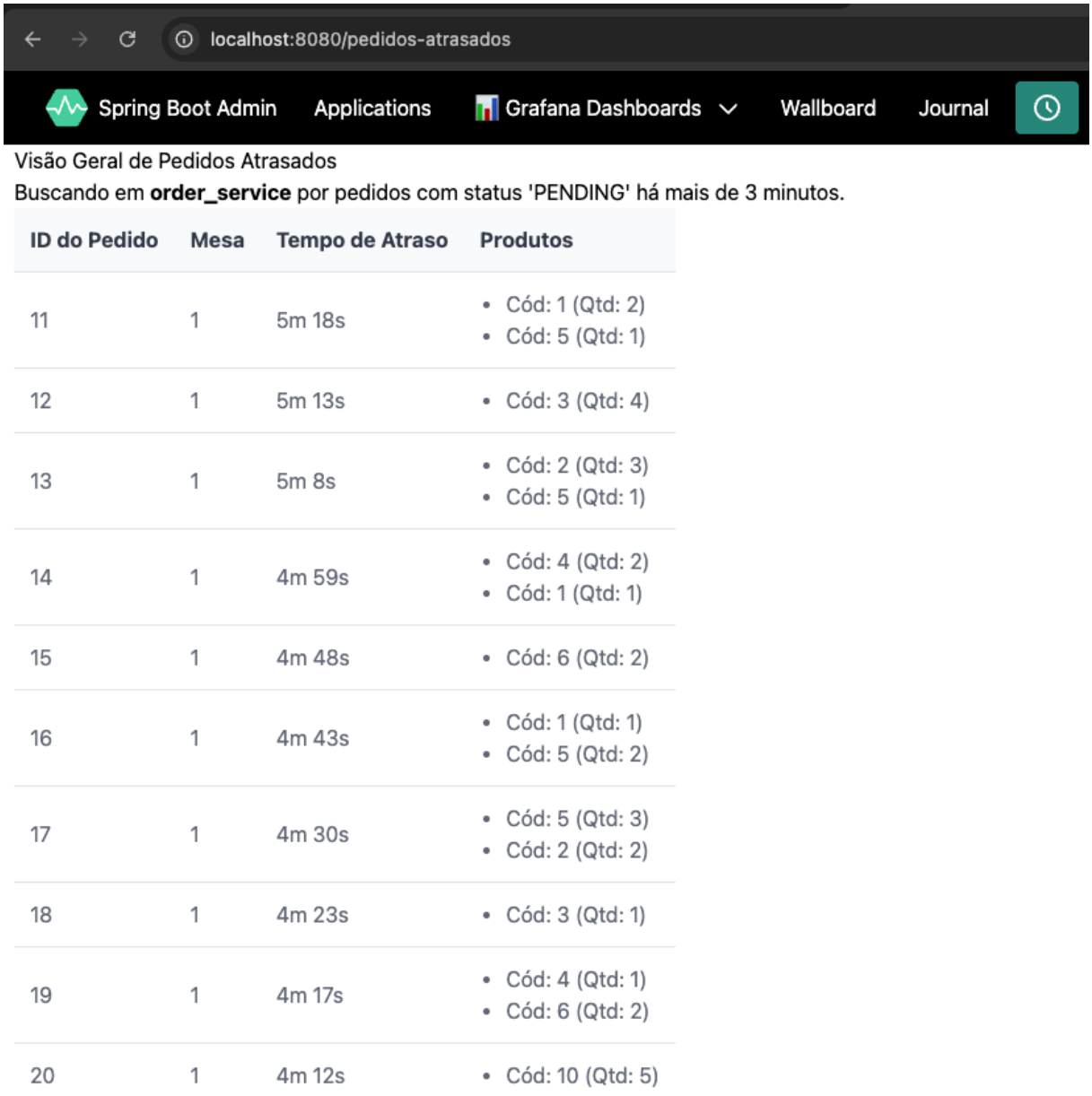

Figura 5.10: Extensão da interface nativa através da nova página 'Pedidos Atrasados'

5.5 Demonstração do Monitoramento de Logs de Negócio no SBA

Além do monitoramento de métricas e do estado de saúde, uma parte essencial da observabilidade de uma aplicação é a capacidade de inspecionar seus logs em tempo real. O Spring Boot Admin, em conjunto com o Spring Boot Actuator, oferece uma solução nativa e centralizada para essa finalidade, que foi utilizada no projeto para monitorar eventos de negócio importantes.

5.5.1 Geração de Logs com SLF4J

Em toda a aplicação, mas com especial importância na classe (OrderProcessingTask), o framework de logging SLF4J (Simple Logging Facade for Java) foi utilizado para registrar eventos significativos. A (OrderProcessingTask), que atua como um ”fiscal de SLAs”, utiliza o logger para dois propósitos principais:

- Logs Informativos (INFO): A cada execução, a tarefa registra que está iniciando a verificação (logger.info(”Verificando pedidos pendentes...”)). Isso gera um ”heartbeat” nos logs, confirmando que a tarefa agendada está viva e operando como esperado.
- Logs de Aviso (WARN): Quando um pedido que viola o SLA é encontrado, um log de aviso de alta visibilidade é gerado (logger.warn(”ATENÇÃO - PEDIDO ATRASADO !!! ...”)). Este log não representa um erro no sistema, mas sim um evento de negócio que requer atenção, como um atraso no processamento.

Listagem 5.1: Trecho da função ”processPendingOrders” onde ocorre o disparo dos logs

```
@Scheduled(fixedRate = 30000)
public void processPendingOrders() {
    logger.info("Verificando␣pedidos␣pendentes␣com␣mais␣de␣3␣
       minutos...");

    LocalDateTime cutoffTime = LocalDateTime.now().minusMinutes
       (3);
    List<Order> delayedOrders = orderService.
       findPendingOrdersOlderThan(cutoffTime);

    for (Order order : delayedOrders) {
      try {
        logger.warn(
            "ATENCAO␣-␣PEDIDO␣ATRASADO!␣Pedido␣ID␣{}␣esta␣
               atrasado␣(criado␣em:␣{}).",
            order.getId(),
            order.getCreatedAt().format(DateTimeFormatter.
               ISO_LOCAL_DATE_TIME)
        );
      } catch (Exception e) {
        logger.error("Erro␣ao␣verificar␣pedido␣ID␣{}:␣{}", order.
           getId(), e.getMessage());
```

```
        }
    ...
    ...
}
```

5.5.2 Visualização Centralizada no Painel do SBA

A validação desta funcionalidade foi realizada diretamente na interface do Spring Boot Admin. Para que o SBA pudesse acessar os logs, duas configurações foram essenciais nos arquivos application.properties dos microsserviços:

1. **management.endpoints.web.exposure.include=***: Garantiu que o *endpoint* /actuator/logfile fosse exposto.
2. **logging.file.name**: Definiu um arquivo de saída para os logs (ex: logs/application.log), pois o Actuator precisa de um arquivo físico para ler.

Com essa configuração, a aba "Logfile" na página de detalhes da instância do order_service no SBA se tornou um console de monitoramento de negócio em tempo real.

Demonstração Prática:

- Ao observar o logfile, foi possível confirmar a execução periódica da OrderProcessingTask através das mensagens de INFO.
- Após a criação de um pedido e a espera de mais de três minutos, a mensagem de WARN sobre o pedido atrasado apareceu claramente no log, destacada pela formatação de cor padrão do SBA.
- O painel também oferece a funcionalidade de download dos logs completos e a capacidade de atualizar a visualização em tempo real, permitindo um acompanhamento contínuo dos eventos da aplicação.

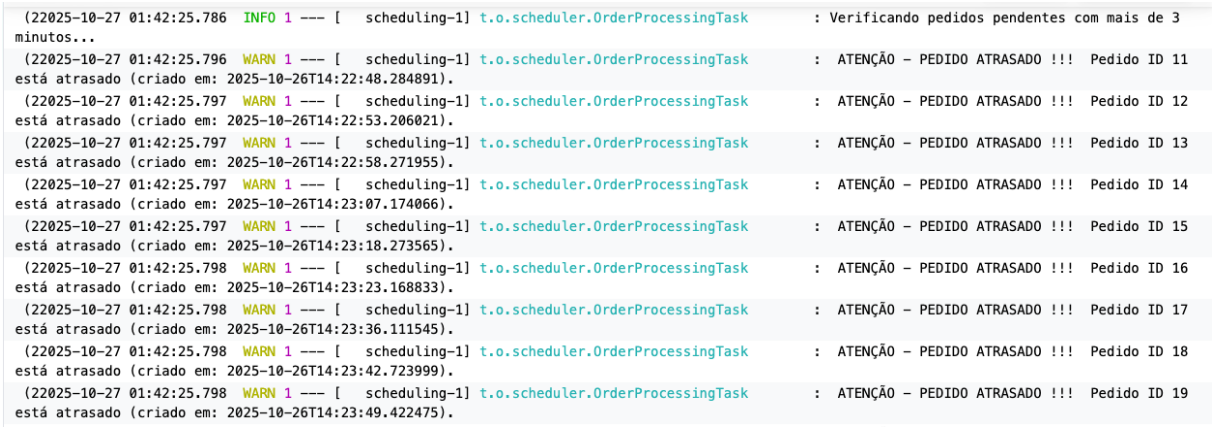

Figura 5.11: Visualização dos logs na interface nativa do SBA

Este resultado demonstra que, mesmo sem ferramentas externas, o uso combinado de um *logging* estruturado na aplicação e da interface nativa do SBA fornece uma poderosa ferramenta para a observabilidade de eventos de negócio, permitindo que um operador identifique e reaja a situações como atrasos no processamento de pedidos diretamente pelo painel de monitoramento.

Capítulo 6

Considerações finais

O presente trabalho teve como objetivo central explorar e aprimorar a observabilidade em um ecossistema de microsserviços, utilizando o Spring Boot Admin (SBA) como ferramenta primária. Através da integração com a pilha Prometheus/Grafana e do desenvolvimento de extensões customizadas, o projeto demonstrou como mitigar as limitações inerentes do SBA e transformá-lo em um hub operacional robusto, capaz de gerar *insights* técnicos e de negócio.

6.1 Resumo dos Resultados Alcançados

Os resultados práticos e demonstrações no Capítulo 5 validaram a eficácia da arquitetura aprimorada, confirmando o cumprimento dos objetivos específicos do trabalho. Os principais resultados alcançados incluem:

- Observabilidade Híbrida e Histórica: A integração do SBA com Prometheus e Grafana superou a limitação do SBA em lidar com análise quantitativa e histórica de métricas. Foi demonstrada a criação de dashboards customizados no Grafana para análise de desempenho (latência e *throughput*) e a análise de tendências de KPIs. Além disso, a interface do SBA foi estendida com "External Views"para fornecer um acesso unificado e contextualizado aos dashboards do Grafana.

- Instrumentação de Inteligência de Negócio: Foi implementada uma instrumentação avançada no ‘order_service‘ utilizando o Micrometer. Isso permitiu a coleta de métricas estratégicas, como o volume total de produtos vendidos (‘order.products.total‘) e, notavelmente, a métrica de combinação de produtos (‘order.product.combinations‘), que possibilita a análise de cesta de compras em tempo real.

- Sistema de Alertas Robusto e Multicanal: O sistema de notificação foi ampliado com sucesso. Foi validado o DiscordNotification, que utiliza ”Embeds” com cores

dinâmicas para alertas visuais. Mais importante, a ausência de suporte nativo ao WhatsApp foi resolvida com o WhatsAppNotification, que utiliza a Twilio como gateway, adaptando-se às exigências de sua API.

- Extensão Operacional e Gerenciamento Ativo: O SBA foi validado como ferramenta de gerenciamento ativo, não apenas de visualização. Foi demonstrada a funcionalidade de gerenciamento da tarefa agendada (OrderProcessingTask), incluindo a visualização do status e a capacidade de acionamento manual ("Trigger"). Adicionalmente, a interface foi estendida com uma Custom View ("Pedidos Atrasados"), provando a capacidade de injetar *insights* de negócio (violação de SLA) de forma nativa na UI. Finalmente, foi confirmada a capacidade de visualização de logs de negócio diretamente na aba "Logfile" do SBA.

6.2 Limitações do Trabalho e Ameaças à Validade

Embora os objetivos centrais tenham sido alcançados, é crucial reconhecer as limitações impostas pelo escopo e pelo ambiente de experimentação:

- Ausência de Rastreamento Distribuído (Tracing): A arquitetura se concentrou em métricas e logs, mas não incorporou uma solução de "Distributed Tracing". Ignorar o rastreamento é considerado um anti-padrão que cria "pontos cegos" na latência *cross-service* [12]. Em outras palavras, uma arquitetura de microsserviços, o *tracing* é fundamental para diagnosticar a latência de ponta a ponta de uma transação que atravessa vários serviços, algo que métricas agregadas (como latência de endpoint HTTP) não conseguem fazer sozinhas.

- Ambiente de Persistência Simplificado: Os serviços de negócio utilizaram o banco de dados H2 em memória. Embora essa escolha tenha simplificado a orquestração com Docker Compose e garantido a reprodutibilidade, ela não representa um ambiente de produção real, onde a persistência de dados ocorre em bancos gerenciados e distribuídos.

- Segurança em Nível Experimental: As configurações de segurança foram adaptadas para um ambiente de testes local, permitindo o acesso irrestrito para facilitar a experimentação. Em um cenário de produção, seria necessário implementar mecanismos de autenticação e autorização muito mais robustos, especialmente para endpoints críticos do Actuator, como '/shutdown' e '/restart'.

6.3 Trabalhos Futuros

Com base nas limitações identificadas e nas tendências atuais de observabilidade, propõem-se os seguintes caminhos para futuros trabalhos, visando aprimorar ainda mais o ecossistema de monitoramento.

6.3.1 Implementação de Rastreamento Distribuído (Distributed Tracing)

O próximo passo lógico na evolução da observabilidade seria a implementação de *Distributed Tracing*.

- Ferramentas Sugeridas: Adoção de padrões como OpenTelemetry (OTEL) para instrumentação, em conjunto com um backend de *tracing* como Jaeger ou Zipkin.
- Valor Agregado: Isso permitiria a visualização completa do caminho que um pedido percorre, desde a requisição inicial no order_service até a validação no product_service. O *tracing* forneceria a correlação necessária entre logs, métricas e rastros, fechando o ciclo dos três pilares da observabilidade.

6.3.2 Aprimoramento da Estratégia de Coleta de Logs

Embora o acesso via /logfile do SBA tenha sido validado para depuração, essa abordagem não é escalável para múltiplos serviços ou instâncias. A agregação centralizada de logs é crucial, pois a observabilidade moderna requer logs, métricas e traces para um entendimento completo do sistema [10, 11].

- Ferramentas Sugeridas: Implementação de um sistema centralizado de agregação de logs. Isso pode ser alcançado com a pilha Loki, que se integra nativamente ao Prometheus e Grafana, ou com a tradicional pilha ELK (Elasticsearch, Logstash, Kibana).
- Valor Agregado: A agregação de logs permitiria a busca e análise em texto completo (*full-text search*) em logs de todas as instâncias simultaneamente, facilitando a correlação de eventos através de identificadores de transação únicos, em vez de inspecionar arquivos de log individuais.

Em conclusão, o trabalho demonstrou que o Spring Boot Admin, quando integrado de forma estratégica e estendido com funcionalidades customizadas, é uma ferramenta poderosa e flexível, capaz de sustentar uma arquitetura de observabilidade moderna e orientada a resultados de negócio.

Referências

[1] Thönes, Johannes: *Microservices.* IEEE Software, 32, 2015. https://ieeexplore.ieee.org/document/7030212. 1, 6

[2] Gomes, Francisco A. A., Vinicius B. Gabriel, Paulo A. L. Rego, Fernando A. M. Trinta e José N. de Souza: *Impact of opentelemetry configuration on observability and telemetry storage cost of microservices-based applications.* Proceedings of the International Conference on Information Technology (ICIT), 2025. https://hal.science/hal-04723959/document. 2

[3] Giamattei, L., A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dînga, A. Koziolek, S. Singh, M. Armbruster, J.M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. Fernandez Vogelin e F. Simon Panojo: *Monitoring tools for devops and microservices: A systematic grey literature review.* Journal of Systems and Software, 208, 2024. https://www.sciencedirect.com/science/article/pii/S0164121223003011. 2, 7

[4] Fei, Jennifer, Jessica Sadye Wolff, Michael Hotard, Hannah Ingham, Saurabh Khanna, Duncan Lawrence, Beza Tesfaye, Jeremy Weinstein, Vasil Yasenov e Jens Hainmueller: *Automated chat application surveys using whatsapp.* Pre-print/Working Paper (ou periódico de submissão), 2020. https://www.researchgate.net/publication/346566143_Automated_Chat_Application_Surveys_Using_WhatsApp. 2, 38

[5] Yulianto, S V, L D Setia e A P Atmaja: *The use of whatsapp gateway for automatic notification system.* Journal of Physics: Conference Series, 2021. https://www.researchgate.net/publication/350336606_The_Use_of_Whatsapp_Gateway_for_Automatic_Notification_System. 2, 38

[6] Amaro, Ricardo, Rúben Pereira e Miguel Mira da Silva: *Devops metrics and kpis: A multivocal literature review.* ACM Computing Surveys, 56, 2024. https://dl.acm.org/doi/full/10.1145/3652508. 3, 7, 8

[7] Ferrer, Borja Ramis, Usman Muhammad, Wael M. Mohammed e José L. Martínez Lastra: *Implementing and visualizing iso 22400 key performance indicators for monitoring discrete manufacturing systems.* Machines, 6, 2018. https://www.mdpi.com/2075-1702/6/3/39. 4, 8

[8] Claus Pahl, Pooyan Jamshidi: *Microservices: A systematic mapping study.* 1, 2016. https://www.scitepress.org/PublishedPapers/2016/57855/57855.pdf. 6

[9] Abgaz, Yalemisew, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley e Paul Clarke: *Decomposition of monolith applications into microservices architectures: A systematic review*. IEEE Transactions on Software Engineering, 49, 2023. https://ieeexplore.ieee.org/document/10160171. 6

[10] Elradi, Mohammed Daffalla: *Prometheus & grafana: A metrics-focused monitoring stack*. Journal of Computer Allied Intelligence, 2025. https://fringeglobal.com/ojs/index.php/jcai/article/view/prometheus-grafana-a-metrics-focused-monitoring-stack. 7, 10, 63

[11] Faseeha, Ummay, Hassan Jamil Syed, Fahad Samad, Sehar Zehra e Hamza Ahmed: *Observability in microservices: An in-depth exploration of frameworks, challenges, and deployment paradigms*. IEEE Access, 2025. https://ieeexplore.ieee.org/document/10967524. 7, 63

[12] Bhosale, Pradeep: *Metrics, logs, and traces: A unified approach to observability in microservices*. Journal of Artificial Intelligence, Machine Learning and Data Science, 2022. https://urfjournals.org/open-access/metrics-logs-and-traces-a-unified-approach-to-observability-in-microservices.pdf. 7, 62

[13] Pappula, Kiran Kumar, Sunil Anasuri e Guru Pramod Rusum: *Building observability into full-stack systems: Metrics that matter*. International Journal of Emerging Research in Engineering and Technology, 2021. https://ijeret.org/index.php/ijeret/article/view/253. 7

[14] Allam, Hitesh: *Metrics that matter: Evolving observability practices for scalable infrastructure*. International Journal of AI, BigData, Computational and Management Studies, 2022. https://ijaibdcms.org/index.php/ijaibdcms/article/view/180. 7

[15] Matcha, Sasibhushana e Er. Niharika Singh: *Microservices architecture: Design patterns, scalability, and inter-service communication strategies*. International Journal of Computer Science and Engineering (IJCSE), 2025. https://www.researchgate.net/publication/393465991_MICROSERVICES_ARCHITECTURE_DESIGN_PATTERNS_SCALABILITY_AND_INTER-_SERVICE_COMMUNICATION_STRATEGIES. 7

[16] Anand, Abhishek: *Design and implementation of a yaml-driven metrics layer framework for standardized kpi delivery in microservices*. Universal Library of Engineering Technology, 2, 2025. https://ulopenaccess.com/ulpages/fulltextULETE?PublishID=ULETE20250204_002. 8, 18

[17] Sinha, Akash Rakesh: *Optimizing microservices development: The role of micro configuration frameworks and monorepo strategies*. International Journal of Core Engineering & Management, 2021. https://www.researchgate.net/publication/390524705_OPTIMIZING_MICROSERVICES_DEVELOPMENT_THE_ROLE_OF_MICRO_CONFIGURATION_FRAMEWORKS_AND_MONOREPO_STRATEGIES. 9

[18] Simanjuntak, Eko e Nico Surantha: *Multiple time series database on microservice architecture for iot-based sleep monitoring system.* Journal of Big Data, 2022. https://www.researchgate.net/publication/365479171_Multiple_time_series_database_on_microservice_architecture_for_IoT-based_sleep_monitoring_system. 11, 13, 18

[19] Acharya, Jigna N. e Anil C. Suthar: *Docker container orchestration management: A review.* Proceedings of the International Conference on Intelligent Vision and Computing (ICIVC 2021), 2022. https://www.researchgate.net/publication/359449395_Docker_Container_Orchestration_Management_A_Review. 14

[20] Henkel, Jordan, Christian Bird, Shuvendu K. Lahiri e Thomas Reps: *Learning from, understanding, and supporting devops artifacts for docker.* 42nd International Conference on Software Engineering (ICSE '20), 2020. https://dl.acm.org/doi/abs/10.1145/3377811.3380406. 16