KAIO VINÍCIUS MORAIS SILVA

DESACOPLAMENTO DO MÓDULO DE PAGAMENTOS DE UM SISTEMA WEB UTILIZANDO A ARQUITETURA DE MICROSSERVIÇOS

LAVRAS – MG

2022

KAIO VINÍCIUS MORAIS SILVA

DESACOPLAMENTO DO MÓDULO DE PAGAMENTOS DE UM SISTEMA WEB UTILIZANDO A ARQUITETURA DE MICROSSERVIÇOS

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. Dr. Neumar Costa Malheiros
Orientador

LAVRAS – MG

2022

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos da Biblioteca Universitária da UFLA

Silva, Kaio Vinícius Morais
Desacoplamento do Módulo de Pagamentos de um Sistema Web utilizando a Arquitetura de Microsserviços / . 2ª ed. rev., atual. e ampl. – Lavras : UFLA, 2022.
42 p. : il.

TCC(Graduação)–Universidade Federal de Lavras, 2022.
Orientador: Prof. Dr. Neumar Costa Malheiros.
Bibliografia.

1. TCC. 2. Monografia. 3. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. Título.

CDD-808.066

KAIO VINÍCIUS MORAIS SILVA

DESACOPLAMENTO DO MÓDULO DE PAGAMENTOS DE UM SISTEMA WEB UTILIZANDO A ARQUITETURA DE MICROSSERVIÇOS

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 09 de Setembro de 2022.

Prof. Dr. Paulo Afonso Parreira Junior UFLA
Prof. Dr. Maurício Ronny de Almeida Souza UFLA

Prof. Dr. Neumar Costa Malheiros
Orientador

LAVRAS – MG
2022

RESUMO

Com o intenso crescimento do mercado de tecnologia e informação, criou-se um cenário amplamente favorável para o surgimento de serviços e soluções digitais, principalmente no ecossistema web, que visam atender as mais diversas necessidades apresentadas pelas empresas em diversos segmentos de produção. Por consequência, conforme o processo de desenvolvimento de aplicações tornou-se mais robusto nos últimos anos, também abriu-se um espaço importante para discutir e avaliar o modo como tais serviços poderiam ser implementados e organizados, de modo que fosse possível criar sistemas distribuídos escaláveis, que fizessem um uso otimizado dos recursos computacionais empregados. Tendo em vista o cenário descrito, o objetivo deste trabalho é relatar as atividades desenvolvidas durante a realização do estágio na empresa Zeester, focadas na manutenção evolutiva de um sistema web da empresa para alterar a arquitetura para o modelo de microsserviços e também incluir novas funcionalidades ao modelo de negócio proposto. Serão apresentadas algumas considerações relativas aos resultados obtidos com a implementação da novo modelo de arquitetura para integração de serviços, as dificuldades experienciadas durante a realização do projeto e a relação dos conhecimentos adquiridos durante a graduação com o contexto prático de desenvolvimento dentro da empresa.

Palavras-chave: Arquitetura de microsserviços. Back-end. API RESTful. Gateway de pagamento. JavaScript.

ABSTRACT

With the intense growth of the technology and information market, a highly favorable scenario has been created for the emergence of digital services and solutions, especially in the web ecosystem, which aim to meet the most diverse needs presented by companies in various production segments. Consequently, as the application development process has become more robust in recent years, an important space has also opened up to discuss and evaluate how such services could be implemented and organized, so that it is possible to create distributed systems that make optimal use of the computing resources employed. In view of the scenario described, the objective of this work is to report the activities developed during the internship at the company Zeester, focused on the evolutionary maintenance of a company's web system to change the architecture to the microservices model and also include new functionalities in their respective business rule. Finally, some final considerations will be presented regarding the results obtained with the implementation of the company's new services architecture, the difficulties experienced during the project, and the relationship of the knowledge acquired during graduation with the practical context of development within the company.

Keywords: Microservices architecture. Back-end. RESTful API. Payment gateway. JavaScript.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

A empresa Zeester[1] possui a plataforma de aprendizado Blesss[2] como seu principal produto desde 2017, sendo esta uma ferramenta focada na disponibilização de vídeos, artigos e livros para um público-alvo interessado em consumir de forma aprofundada conteúdos ligados a fé cristã. Em seu surgimento, o serviço em questão foi inicialmente desenvolvido tendo como base um modelo de arquitetura monolítica, na qual os principais componentes funcionais foram acoplados em uma única aplicação *back-end*[3] de software. Consequentemente, esse serviço englobou não somente as principais funcionalidades do modelo de negócio mas também outros processos essenciais para o funcionamento do produto, com destaque para o gerenciamento de transações financeiras realizadas pelos usuários dentro da plataforma.

Embora a arquitetura monolítica tenha simplificado o processo de desenvolvimento do produto durante as etapas iniciais do projeto, a plataforma passou por várias atualizações desde o seu surgimento que incorporaram um grande volume de funcionalidades ao serviço, e que também contribuíram para um aumento significativo da complexidade do código-fonte da API[4] principal. Em razão disso, o processo de manutenção tornou-se cada vez mais custoso, além do aparecimento de diversas outras deficiências técnicas que passaram a dificultar a inserção de novas funcionalidades à plataforma.

Portanto, a realização deste projeto representa uma tentativa, por parte da empresa, de reduzir a complexidade de seu sistema ao desacoplar as principais funcionalidades do produto principal. Para isso, o primeiro passo tomado referiu-se ao desacoplamento do mecanismo de processamento de pagamentos. O objetivo dessa decisão era criar uma arquitetura de microsserviços escalável, cujos componentes possam ser utilizados não somente pela plataforma Blesss, mas também por qualquer outro tipo de serviço que a empresa venha a criar em um futuro próximo.

Esta primeira etapa de reestruturação técnica representa o tema principal deste trabalho, que apresenta um relatório sobre a realização de um estágio pela empresa Zeester, referente as principais atividades realizadas pelo estagiário que viabilizaram a implantação dessa nova arquitetura de serviços. O estágio foi realizado no período entre outubro de 2019 até dezembro de 2020, e as tarefas executadas foram focadas na implementação de uma arquitetura de

[1] https://zeester.com.br/
[2] https://blesss.org/
[3] https://www.geeksforgeeks.org/frontend-vs-backend/
[4] https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces

microsserviços, visando o aprimoramento da infraestrutura técnica das aplicações, em razão do surgimento de novos requisitos do modelo de negócio da empresa.

1.1 Ambiente de Trabalho

A Zeester é uma empresa privada especializada em consultoria de tecnologia e desenvolvimento de soluções digitais para negócios de empresas terceiras, com sede oficial na cidade de Lavras-MG. Atualmente, a empresa possui um corpo de aproximadamente 20 funcionários, dividido entre equipes de desenvolvimento, marketing, comunicação e liderança administrativa. Seu principal produto é a plataforma Blesss, uma aplicação especializada na distribuição de conteúdo cristão, e que possui um extenso acervo de vídeos, séries, artigos e livros, além de também realizar transmissões ao vivo de eventos associados ao serviço; a plataforma possui versões para os ecossistemas Android, Web, TVs LG e AndroidTV.

Durante o período de atuação dentro da empresa, o estagiário participou da equipe de desenvolvimento técnico, composta por três desenvolvedores *back-end* e outros três desenvolvedores *front-end*, sendo que um desses integrantes também atuou como líder técnico da equipe. Para estruturar o plano de desenvolvimento e manutenção dos produtos da empresa, a equipe utiliza os principais conceitos da metodologia ágil Scrum (SOARES, 2004) para organizar o plano de trabalho, estimar as dificuldades das tarefas, e separar as etapas de desenvolvimento em *sprints* semanais. Essa abordagem de trabalho permitia a realização de um desenvolvimento iterativo, focado na entrega contínua de resultados para garantir a adição de novas funcionalidades e a manutenção técnica dos serviços existentes oferecidos pela empresa.

1.2 Organização do Documento

Este relatório está organizado como descrito a seguir. O Capítulo 2 apresenta um referencial teórico, contendo todos os conceitos técnicos assimilados e ferramentas tecnológicas utilizadas nas tarefas executadas durante a realização do estágio. O Capítulo 3 descreve as atividades realizadas, e apresentar um panorama geral sobre a arquitetura de microsserviços implementada, com base nas soluções técnicas empregadas. Por fim, o Capítulo 4 apresenta as considerações finais referentes aos resultados obtidos na realização do projeto, e também sobre o aprendizado obtido nas atividades executadas no estágio.

2 REFERENCIAL TEÓRICO

Neste capítulo, são explicados os principais conceitos, abordagens e tecnologias que constituíram a base de conhecimento essencial para o cumprimento das tarefas alocadas durante a realização do estágio. Na Seção 2.1, são apresentados conceitos básicos sobre os principais modelos de arquitetura de aplicações distribuídas. Em seguida, na Seção 2.2, é apresentada a definição do estilo arquitetural REST. Por fim, a Seção 2.3 apresenta as principais concepções referentes às tecnologias utilizadas neste projeto.

2.1 Modelos de Arquitetura

Nesta seção, são apresentados os principais modelos de arquitetura implementados nos sistemas computacionais atuais. Na Seção 2.1.1, são abordadas as principais definições relativas ao modelo de arquitetura monolítica. A Seção 2.1.2 apresenta os conceitos fundamentais do modelo de arquitetura de microsserviços, seguido pela descrição do modelo de arquitetura orientada a eventos na Seção 2.1.3. Por fim, a Seção 2.1.4 apresenta as principais características do modelo de arquitetura *peer-to-peer*.

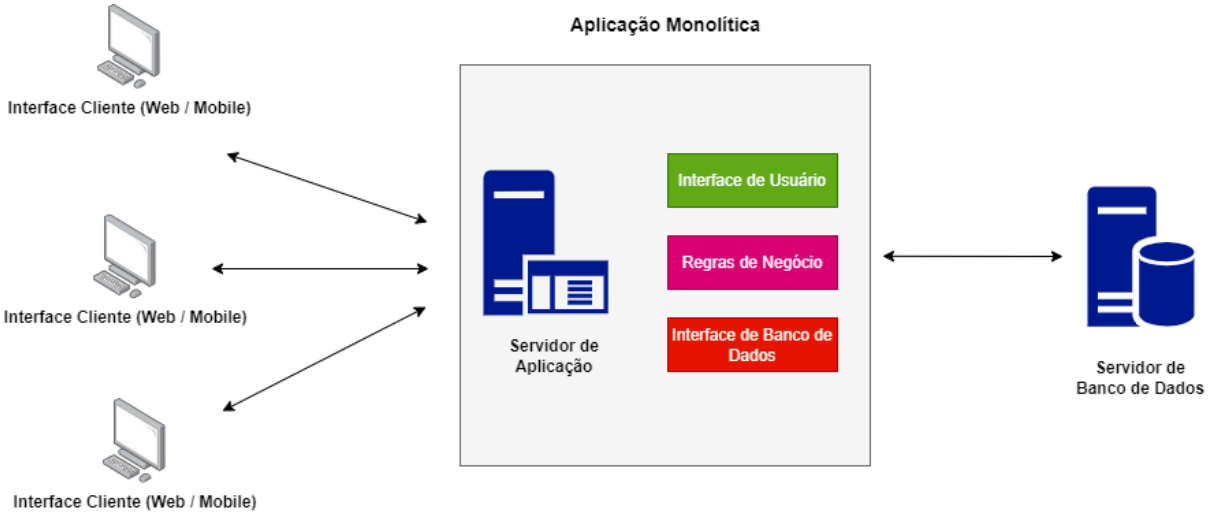
2.1.1 Modelo de Arquitetura Monolítica

Conforme apresentado em (LUCIO et al., 2017), o modelo de arquitetura monolítica representa uma proposta de organização na qual uma única aplicação de software concentra todos os tipos de processos existentes dentro de sua regra de negócio. Consequentemente, tal aplicação pode apresentar a interface de interação com o usuário, acessar e manipular informações persistidas no banco de dados, processar solicitações de seus clientes, além de quaisquer outras funcionalidades que a aplicação seja responsável por processar.

Dentro do ecossistema de serviços web, uma aplicação monolítica pode ser representada por uma arquitetura cliente/servidor de três camadas, que envolve um dispositivo cliente, uma aplicação servidora e um servidor de banco de dados (OLUWATOSIN, 2014). Neste modelo organizacional, conforme demonstrado pela Figura 2.1 , um respectivo cliente da aplicação pode realizar solicitações para o serviço em questão utilizando um protocolo de comunicação adequado; em seguida, a aplicação responsabiliza-se por processar todas as operações lógicas necessárias para satisfazer a requisição de seu cliente, incluindo a execução de todos os proce-

dimentos requeridos pela regra de negócio do serviço e o processamento da leitura e escrita de registros no banco de dados utilizado.

Figura 2.1 – Exemplo de uma arquitetura cliente/servidor



Fonte: Do autor

Este modelo arquitetural, quando utilizado corretamente, pode trazer alguns benefícios relevantes. O processo de desenvolvimento inicial de uma aplicação, por exemplo, pode ser agilizado devido à simplicidade de desenvolver um único código-fonte que contenha todas as funcionalidades necessárias para a regra de negócio. A realização de testes *end-to-end* sobre as funcionalidades também apresentam ganhos de performance em comparação a uma arquitetura de serviços distribuídos, considerando que os módulos funcionais estão centralizados em uma única aplicação. Esses tipos de testes são importantes para avaliar o comportamento e o funcionamento do sistema considerando a perspectiva de um usuário final do serviço (TSAI et al., 2001). Em razão disso, o processo de depuração para localizar erros na aplicação também torna-se menos complexo e mais ágil em comparação a outros modelos arquiteturais (ATLASSIAN, 2022).

A implementação de aplicações monolíticas, no entanto, também possui desvantagens importantes e que devem ser levadas em consideração quando da concepção da arquitetura da solução. Um sistema monolítico pode apresentar um nível indesejável de escalabilidade, considerando a impossibilidade de escalar um componente individual da aplicação sem que haja a necessidade de replicá-la como um todo (MENARD, 2020). Conforme o sistema cresce e incorpora novas funcionalidades, torna-se cada vez mais difícil realizar adaptações na aplicação, em razão de sua alta complexidade (TAPIA et al., 2020). Consequentemente, a ocorrência de qualquer erro dentro de algum módulo integrado pode comprometer o funcionamento de todo

o serviço. A flexibilidade para adotar novas tecnologias também é restringida, pois qualquer mudança na aplicação consistiria em um processo de atualização mais lento e custoso para garantir a integridade e a disponibilidade do sistema (ATLASSIAN, 2022).

Com o passar dos anos, as aplicações ficaram mais complexas e passaram a lidar com maiores desafios técnicos para manter níveis aceitáveis de robustez. A integração de ferramentas e componentes como *brokers*, *proxies* e bancos de dados, por exemplo, representam requisitos cada vez mais importantes para viabilizar o bom funcionamento dos serviços existentes. O surgimento de novas tecnologias e a popularização da computação em nuvem também são aspectos relevantes que passaram a influenciar diretamente o modo como os sistemas são desenvolvidos. Em função disso, modelos arquiteturais mais modernos surgiram, e novas reflexões passaram a ser feitas sobre a relação entre produtividade de desenvolvimento e complexidade técnica apresentada pela arquitetura monolítica em comparação a outros modelos existentes (SOFTWARE, 2021), de modo a contestar sua ampla adoção em sistemas contemporâneos.

Contudo, ainda que novos modelos de arquitetura tenham surgido nos últimos anos para atender uma maior gama de necessidades técnicas, isso não necessariamente implica que a arquitetura monolítica tenha perdido sua relevância. Esse modelo ainda pode ser adequado para o desenvolvimento de aplicações menos complexas, que não incorporam um volume muito grande de funcionalidades; esses tipos de serviços podem assumir uma característica de autosuficiência ao agrupar todas as responsabilidades e funções em um único processo (LUCIO et al., 2017), garantindo assim o usufruto das vantagens oferecidas por esse modelo arquitetural.

2.1.2 Modelo de Arquitetura de Microsserviços

O modelo de arquitetura de microsserviços pode ser definido como uma abordagem arquitetural focada no desenvolvimento de uma única aplicação como um conjunto de pequenos serviços modularizados, com cada um executando seu próprio processo e se comunicando por meio de mecanismos leves (FOWLER, 2014). Em função disso, tais serviços são desenvolvidos conforme as necessidades da respectiva regra de negócio, e podem ser implantados de forma independente por máquinas de implantação totalmente automatizadas. Consequentemente, esse estilo arquitetural preza pelo mínimo gerenciamento centralizado desses serviços, que podem ser desenvolvidos utilizando diferentes linguagens de programação e tecnologias de armazenamento de dados (FOWLER, 2014).

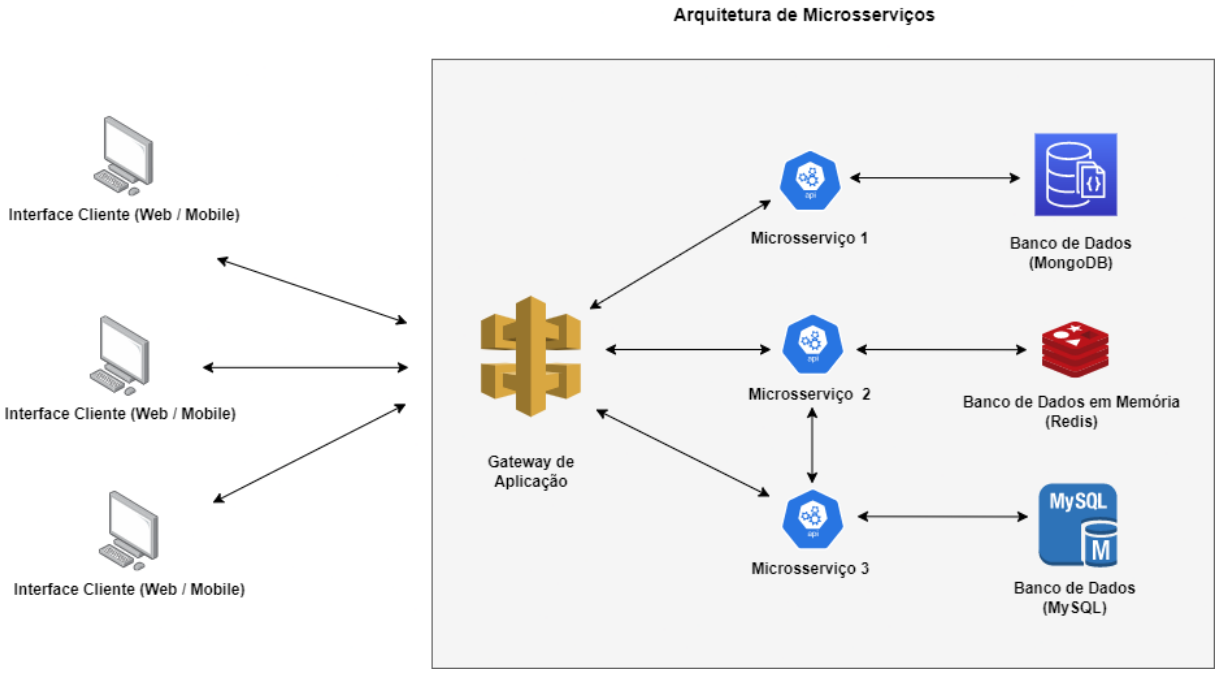
Em razão das características do seu modelo estrutural, a arquitetura de microsserviços pressupõe a divisão das principais funcionalidades do sistema em serviços separados, o que facilita o gerenciamento do sistema ao dividir as tarefas em módulos funcionais menores e independentes. Isso contribui para que seja possível aplicar atualizações de software e adicionar novas funcionalidades ao sistema com mais frequência e com maior confiabilidade, sem comprometer a integridade dos outros serviços (ATLASSIAN, 2022). Esse paradigma arquitetural também possibilita que o sistema como um todo seja mais escalável, e que ele consiga ajustar seu desempenho funcional em razão da carga de trabalho recebida.

A Figura 2.2 demonstra um exemplo de uma aplicação baseada em uma arquitetura de microsserviços. Nesta representação, as requisições HTTP advindas da interface cliente são gerenciadas por um *gateway* de API. Esse tipo de *gateway* atua como uma espécie de *proxy* reverso[1], capaz de rotear solicitações para os microsserviços *back-end* adequados, com base nos tipos de recursos solicitados por cada cliente da aplicação (MICROSOFT, 2022). Um *proxy* reverso responsabiliza-se por gerenciar o tráfego de requisições de clientes, encaminhar as solicitações para a aplicação servidora desejada, e retornar a resposta do servidor para o cliente, sem que este esteja ciente da estrutura interna do sistema (SOMMERLAD, 2003). Cada microsserviço representa um componente autônomo, responsável por realizar uma tarefa específica, além de estar apto a possuir um banco de dados individual. Essa separação de responsabilidades colabora para que a aplicação como um todo seja mais resiliente, com maior grau de tolerância a falhas. Em função dessas características, esse modelo de arquitetura também viabiliza o encapsulamento da estrutura de serviços da aplicação, possibilitando que seus usuários possam usufruir das funcionalidades ofertadas sem terem conhecimento de como esses componentes estão estruturados internamente.

A arquitetura de microsserviços não representa um modelo técnico perfeito, mas apresenta diversas vantagens que fazem com que muitas empresas do segmento tecnológico adotem essa abordagem como um meio de implementar serviços escaláveis. O desenvolvimento de novas funcionalidades para uma aplicação, por exemplo, torna-se mais ágil, e a capacidade de implantação de novos recursos ao sistema é amplificada. Em razão disso, atualizações em determinados microsserviços podem ser processadas sem que haja o risco de comprometer o funcionamento do sistema como um todo (ATLASSIAN, 2022). O processo de manutenção da infraestrutura também é otimizado, pois a divisão de responsabilidades para cada microsserviço

[1] https://www.nginx.com/resources/glossary/reverse-proxy-vs-load-balancer/

Figura 2.2 – Exemplo de uma arquitetura de microsserviços



Fonte: Do autor

facilita o mapeamento e a correção de débitos técnicos, além de tornar o sistema mais tolerante a falhas (TAPIA et al., 2020).

Contudo, o processo de implantação de uma arquitetura de microsserviços também deve considerar possíveis desvantagens que possam implicar em novas adversidades para as empresas que venham a adotar esse modelo arquitetural. Conforme a infraestrutura interna de serviços cresce e absorve mais requisitos de regra de negócio, o gerenciamento dos microsserviços também torna-se mais complexo, o que dificulta a administração das funcionalidades oferecidas pela aplicação (ATLASSIAN, 2022). Consequentemente, a coordenação de uma organização hierárquica de serviços e a definição dos métodos de comunicação a serem adotados entre eles passam a ser um dos grandes desafios enfrentados na adoção dessa arquitetura. O processo de desenvolvimento também pode se tornar mais custoso, pois cada microsserviço pode assimilar custos individuais de implantação, infraestrutura, hospedagem, ferramentas de monitoramento, e times de suporte técnico especializado (SOFTWARE, 2021).

Sendo assim, a adoção desse modelo de arquitetura representa uma decisão estratégica que precisa considerar todas as características e limitações oferecidas por essa abordagem. A utilização de microsserviços por grandes empresas do segmento tecnológico vem demonstrando como esse paradigma arquitetural pode ser eficaz para lidar com a complexidade de grandes aplicações, considerando ainda a crescente necessidade de prover sistemas escaláveis e alta-

mente confiáveis ao usuário final. Portanto, a escolha pela arquitetura de microsserviços pode ser muito interessante para empresas que trabalham com aplicações mais complexas, que lidam com adições recorrentes de funcionalidades, ou buscam melhorar a escalabilidade e resiliência de seus sistemas.

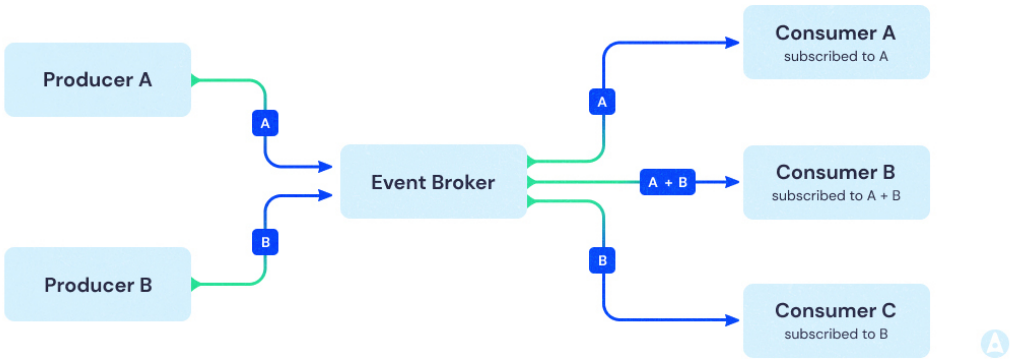
2.1.3 Modelo de Arquitetura Orientada a Eventos

A arquitetura orientada a eventos (*event-driven architecture*) representa um tipo de paradigma arquitetural focado na implementação de uma estrutura de serviços desacoplados, capazes de se comunicarem através da emissão de eventos (SERVICES, 2022). Um evento remete a uma mudança de estado ou uma atualização referente a distintos recursos manipulados dentro ou fora do domínio da aplicação. Cada evento emitido pode conter um corpo de dados para descrever a ocorrência e um cabeçalho contendo informações sobre o tipo do evento emitido, um identificador único, data de emissão, identificador do emissor, além de outros tipos de dados associados. Em função dessas características, a emissão de um evento pode representar um diagnóstico de um problema, o lançamento de um erro, uma indicação da completude de um procedimento, dentre outros tipos de situações possíveis dentro de um sistema (MICHELSON, 2006).

Uma arquitetura orientada a eventos é constituída por três tipos de componentes principais: produtor de eventos, roteador de eventos (*event broker*) e consumidor de eventos (SERVICES, 2022). Nesse cenário, um produtor responsabiliza-se por publicar um evento e repassá-lo a um serviço de roteamento; este, por sua vez, encarrega-se de filtrar os eventos recebidos e enviar tais mensagens aos respectivos consumidores. Consequentemente, este modelo organizacional permite que os processos produtores e consumidores sejam desacoplados, o que facilita a manutenção corretiva e evolutiva dos mesmos. A Figura 2.3 demonstra um exemplo de uma arquitetura orientada a eventos, exemplificando o fluxo de comunicação entre todos os processos componentes do sistema.

Em razão de suas características, uma arquitetura orientada a eventos pode processar quaisquer tipos de eventos que possam estar relacionados com a regra de negócio da aplicação, disseminando seus envios para os processos interessados (MICHELSON, 2006). Portanto, tais processos responsabilizam-se por avaliar o evento recebido e realizar as ações programadas para tratamento dos respectivos eventos. Neste contexto, o produtor não necessariamente precisa conhecer seu consumidor, ou ser informado do resultado do processamento do evento, o que

Figura 2.3 – Exemplo de uma arquitetura orientada a eventos



Fonte: (ADSERVIO, 2022)

caracteriza um estilo de comunicação assíncrona. Ainda conforme explicado em (REDHAT, 2019), a arquitetura orientada a eventos pode ser baseada em dois tipos de modelos:

- **Modelo Pub/Sub:** Esse modelo remete a uma infraestrutura de mensageria baseada em subscrições em um fluxo de eventos. Por conseguinte, após a publicação de um evento, o *broker* deve reconhecer e encaminhar esse evento aos processos subscritores.
- **Modelo de fluxo de eventos:** Neste modelo arquitetural, cada evento publicado é inserido em uma fila de logs, na qual as aplicações consumidoras possuem a liberdade de acessar quaisquer eventos registrados na fila, sem necessariamente serem processos subscritores desse tópico.

A implantação desse paradigma arquitetural pode trazer um interessante conjunto de benefícios. Em razão de suas características naturais, uma arquitetura orientada a eventos é altamente distribuída e possui um baixo nível de acoplamento (REDHAT, 2019), além de agregar vários aspectos técnicos também apresentados pela arquitetura de microsserviços. Esse modelo também permite que um produtor de eventos emita eventos para múltiplos serviços consumidores (ETZION, 2005), sendo este um recurso extremamente interessante para a construção de sistemas distribuídos. Além disso, também destaca-se que, devido ao fato dos componentes produtores não terem conhecimento do processamento posterior do evento ou de seus serviços consumidores, a implementação dessa arquitetura torna-se bastante atraente para sistemas que necessitam lidar com fluxos de operações assíncronas ou com o processamento paralelo de informações.

2.1.4 Modelo de Arquitetura *Peer-to-Peer*

O modelo de arquitetura *peer-to-peer* (P2P) representa um paradigma arquitetural especializado na construção de sistemas computacionais distribuídos com uma estrutura descentralizada e autônoma, sem necessariamente possuir uma restrição sobre o número de componentes integrantes da arquitetura, ou qualquer tipo de diferenciação entre os nós componentes da estrutura (RAMOS et al., 2009). Em função de suas características essenciais, principalmente por sua descentralização, as arquiteturas P2P possuem a tendência de não adotarem quaisquer tipos de restrições ou regulamentações advindas de autoridades centralizadoras (AGRE, 2003).

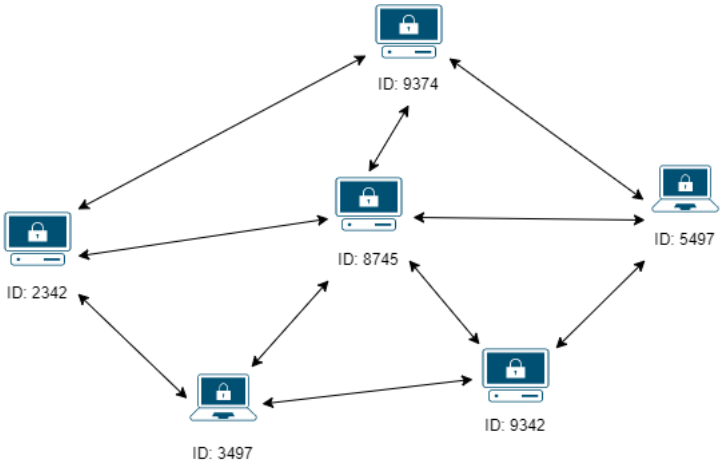
A arquitetura *peer-to-peer* pode ser caracterizada como uma rede colaborativa, focada primariamente no compartilhamento e distribuição de dados e arquivos, permitindo o acesso a recursos agregados e geograficamente distribuídos. O custo para a criação desse tipo de estrutura, no entanto, não necessariamente implica em um alto investimento em equipamentos (RIGHI; PELLISSARI; WESTPHALL, 2004). Em função de sua organização descentralizada, os sistemas *peer-to-peer* apresentam um alto nível de escalabilidade, pois conseguem lidar com o crescimento ou a diminuição do número de nós componentes da arquitetura para manter a rede operacional, além de otimizar o compartilhamento de dados em função da quantidade de unidades participantes. Sua estrutura distribuída também apresenta uma maior robustez contra falhas ou ataques intencionais, considerando que não existe um nó ou um componente de processamento centralizado que possa ser considerado um ponto único de falha da rede.

A Figura 2.4 apresenta um exemplo básico de como uma rede P2P pode ser construída, contendo vários nós conectados e focados no compartilhamento de informações entre todos os integrantes do sistema. Neste modelo, não há diferenciação entre os nós do sistema em termos funcionais. Todos os nós podem enviar ou requisitar dados uns dos outros.

Conforme apresentado em (RAMOS et al., 2009), para que uma estrutura de serviços possa ser considerada uma rede *peer-to-peer*, a mesma precisa apresentar um conjunto de características essenciais que são requisitos mínimos para a construção de uma arquitetura P2P. Dentre tais requisitos, destacam-se os seguintes:

- Um nó componente da rede pode ter conectividade temporária e endereçamento variável.
- A rede precisa estar apta para lidar com diferentes taxas de transferência de dados entre os nós computacionais.

Figura 2.4 – Exemplo de uma arquitetura *peer-to-peer*



Fonte: Do autor

- O sistema precisa garantir a conectividade dos nós contidos nas bordas da rede, para que possam se comunicar com os demais integrantes.
- Os nós podem ter uma autonomia parcial ou total em relação a uma unidade servidora centralizada.
- A rede de nós precisa ser escalável.
- Cada nó deve ser capaz de comunicar-se diretamente com os outros nós componentes da rede.
- A rede precisa garantir que cada nó seja capaz de consumir e fornecer recursos aos outros nós de forma equivalente aos demais.

Existem algumas variações do modelo *peer-to-peer* que usufruem de suas características fundamentais, mas que também combinam outros aspectos técnicos vistos em outros tipos de arquitetura; o modelo híbrido, por exemplo, representa uma versão adaptada da arquitetura P2P na qual existe um ponto centralizador responsável por catalogar e indexar os dados transmitidos entre os membros da rede. Nesse modelo organizacional, um nó participante pode interagir com um servidor central para obter uma informação específica, e receber uma resposta que permita identificar quais componentes da rede possuem o recurso desejado, visando estabelecer uma comunicação direta com o mesmo (RIGHI; PELLISSARI; WESTPHALL, 2004). A adição de um componente centralizador pode trazer uma diminuição na complexidade arquitetural do sistema e simplificar o processamento P2P entre os componentes da rede. No entanto, sua

inserção pode estabelecer um ponto único de falha no sistema, impossibilitando que a rede funcione sem a participação do nó central.

O modelo puro também representa uma das propostas existentes para a arquitetura peer-to-peer, fortemente baseada em uma abordagem de computação colaborativa, na qual os componentes participantes da rede processam comunicações diretas com os outros nós participantes; por consequência, não somente a transferência de informações é compartilhada, mas também qualquer tipo de processamento computacional que envolve a tomada de decisões sobre o funcionamento e a coordenação da rede distribuída (RIGHI; PELLISSARI; WESTPHALL, 2004).

2.2 Estilo Arquitetural REST

O modelo REST (*Representational State Transfer*) pode ser definido como um conjunto arquitetural de restrições que determinam como sistemas distribuídos podem se comunicar a partir de protocolos comumente utilizados em ecossistemas web (REDHAT, 2020). Portanto, qualquer serviço que se comporte em conformidade com as diretrizes impostas pela arquitetura REST pode ser considerada um serviço RESTful.

A arquitetura REST, originalmente, foi criada com o intuito de usufruir das diversas características do protocolo HTTP, sendo essa uma grande vantagem apresentada por este estilo arquitetural. Em razão de tais peculiaridades, existe uma quantidade expressiva de sistemas que também adotam o HTTP como um protocolo de comunicação padrão, e que também usufruem do mesmo conjunto de restrições e benefícios oferecidos por esse modelo. Por consequência, essa ampla adoção contribui diretamente para o aumento gradativo da escalabilidade e da longevidade dos serviços baseados nessa arquitetura (BIEHL, 2016).

Portanto, para que serviço web possa ser considerada do tipo RESTful, ele precisa apresentar um conjunto de características essenciais, referentes ao modelo REST. Um requisito fundamental remete à necessidade do sistema estar inserido em uma arquitetura cliente/servidor, embasada na troca de recursos entre aplicações clientes e servidoras, com solicitações gerenciadas pelo protocolo HTTP. Um outro requisito essencial é a exigência de que a interação entre cliente e servidor seja uma comunicação *stateless*; ou seja, as solicitações HTTP representam requisições isoladas entre si, sem qualquer armazenamento de informações ou estados de sessões referentes a outras solicitações já realizadas (REDHAT, 2020).

Além do seu conjunto de restrições, a arquitetura REST também trabalha com o conceito de gerenciar recursos como seu principal objetivo. Ainda conforme apresentado em (BIEHL,

2016), um recurso pode ser definido como um modelo de dados abstrato responsável por representar inúmeros tipos de informações, que devem ser manipuladas em algum formato textual (XML, JSON, dentre outros tipos de representações), e que pode ser identificado de forma única por sua URI (Uniform Resource Identifier). Portanto, para que seja possível realizar uma requisição utilizando a respectiva URI do recurso visado, também é necessário prover o método HTTP a ser utilizado na solicitação a ser processada. Dentre os principais métodos fornecidos pelo protocolo HTTP, destacam-se:

- **GET:** Método utilizado para solicitar uma representação de dados de um determinado recurso.
- **POST:** Método responsável por submeter os dados necessários para criar um novo recurso.
- **PUT:** Método responsável por processar uma atualização de um recurso pré-existente com o conjunto de dados fornecido pela requisição.
- **PATCH:** Método utilizado para processar uma atualização parcial de um recurso pré-existente com os dados fornecidos pela requisição.
- **DELETE:** Método responsável por realizar a exclusão de um determinado recurso.

Portanto, a combinação de uma URI do recurso desejado, um método HTTP, um conjunto de cabeçalhos e um corpo de dados anexado configuram uma requisição em conformidade com as exigências da arquitetura REST e do protocolo HTTP. Consequentemente, o gerenciamento de estado de cada recurso consultado ou manipulado é realizado pelo processamento das requisições feitas por uma aplicação cliente a uma API responsável por servir tais informações. Em outros termos, é responsabilidade da API manipular o estado de cada recurso requerido, conforme as solicitações recebidas pelo processo cliente (BIEHL, 2016).

2.3 Tecnologias

Nesta seção, são apresentados os principais conceitos e definições referentes às tecnologias utilizadas no projeto para o desenvolvimento dos serviços componentes da plataforma Blesss.

2.3.1 Node.js

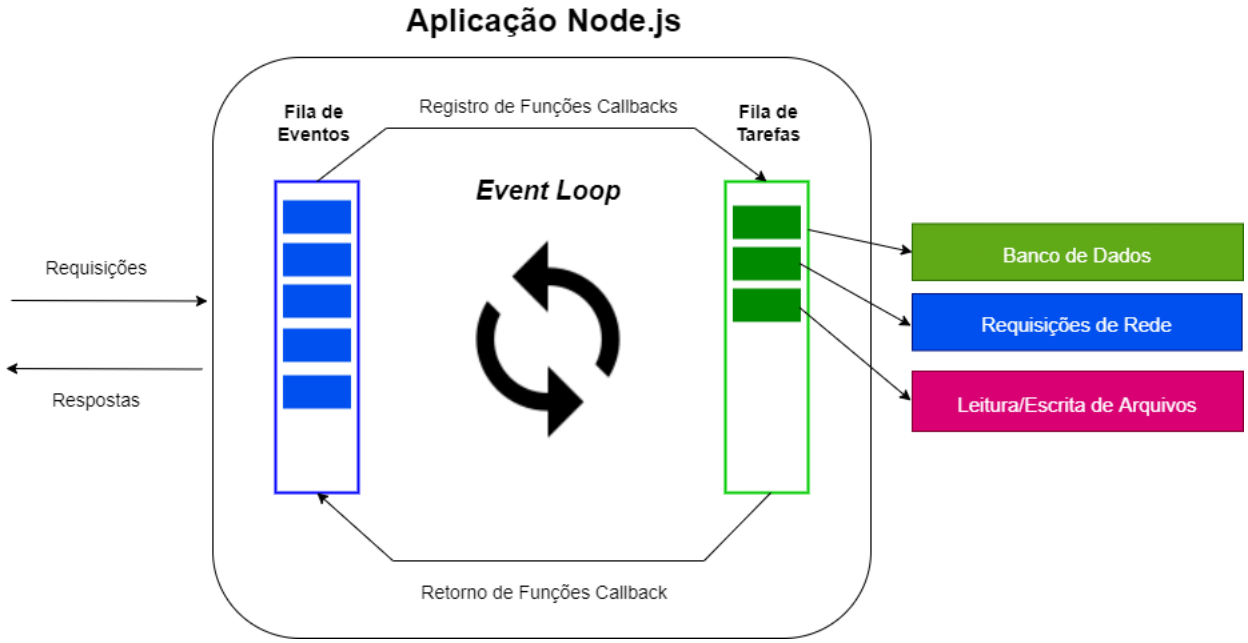
Conforme explicado em (OLIVEIRA; ZANETTI, 2021), o Node.js [2] representa um ambiente de servidor de código aberto, que viabiliza a criação e execução de aplicações *back-end* utilizando a linguagem de programação JavaScript[3]. Para isso, tal tecnologia usufrui do Chrome V8 Engine para viabilizar o processo de interpretação e execução de códigos desenvolvidos em JavaScript visando a construção de aplicações *server-side*.

Criado em 2009 por Ryan Dahl e sua equipe de colaboradores, o Node.js surgiu com uma característica fundamental que o diferencia de outras tecnologias que já estavam presentes no mercado: o modelo de arquitetura apresentado pelo mesmo baseia-se numa abordagem *non-blocking thread*, na qual, para cada aplicação, existe uma única *thread* responsável por tratar as requisições recebidas como uma fila de eventos. Essa estratégia viabiliza o processamento de tais tarefas de forma assíncrona, evitando a ocorrência de *dead-locks* ou sobrecarga de requisições em razão do gerenciamento paralelo de múltiplas *threads* (PEREIRA, 2014). Consequentemente, o Node.js surge como uma boa alternativa para a implementação de sistemas que trabalham mais intensamente com operações de I/O, tendo em vista o bom desempenho entregue com um aproveitamento mais eficiente dos recursos de *hardware* disponíveis para o desenvolvimento da aplicação desejada (BANGARE et al., 2016).

Para viabilizar a arquitetura proposta, o Node.js trabalha com a implementação de um *Event Loop*, uma estrutura orientada a eventos responsável por gerir todas as requisições ou operações solicitadas para a aplicação (LORING; MARRON; LEIJEN, 2017). Conforme demonstrado pela Figura 2.5, o componente consiste numa estrutura de *loop* infinito, na qual, a cada iteração, verifica-se se um determinado evento foi emitido pela aplicação. Caso a emissão ocorra, a operação é inserida em uma lista de tarefas assíncronas a serem executadas pela aplicação. Após a inserção da tarefa, a estrutura de *loop* realiza verificações iterativas para consultar o status de processamento da tarefa, até que a mesma seja finalizada. Ao ser processada, a tarefa retorna uma *callback*, uma função responsável por realizar um determinado procedimento assim que a operação postulada tenha sido concluída. Com o retorno da função *callback*, o processamento é concluído com o tratamento dos dados retornados pela operação de I/O para completar a solicitação realizada.

[2] https://nodejs.org/pt-br/
[3] https://developer.mozilla.org/pt-BR/docs/Web/JavaScript

Figura 2.5 – Funcionamento do Event Loop



Fonte: Do autor, adaptada de (GEEKSFORGEEKS, 2021)

2.3.2 MongoDB

O MongoDB [4] é um banco de dados não relacional (NoSQL), orientado a documentos, voltado para o desenvolvimento ágil de aplicações web. Ele utiliza uma modelagem de dados padronizada, focada em uma alta vazão de operações de leitura e escrita de registros (BANKER et al., 2016). Em comparação com bancos de dados relacionais, o MongoDB oferece a possibilidade de criar modelos de dados com estruturas hierárquicas ricas e mais intuitivas, sendo esta uma grande vantagem a ser considerada. O formato padrão de documentos baseia-se numa formatação de dados em JSON (*JavaScript Object Notation*), um tipo de representação de dados bastante utilizado por outras tecnologias pertencentes ao ecossistema de serviços web. O armazenamento dos dados, no entanto, usufrui do tipo BSON (*Binary JSON*) para a serialização dos documentos em um formato binário, que também viabiliza o processamento de consultas especializadas nos documentos de uma respectiva coleção (MONGODB, 2022).

Para realizar uma comparação com bancos de dados relacionais, uma coleção do MongoDB assemelha-se a uma tabela do modelo relacional, enquanto cada documento criado equivale a um registro armazenado na respectiva tabela (NASCIMENTO; UNIFACCAMP, 2014). Em função de sua estrutura, o processamento de consultas no MongoDB requer custos computacionais menores quando comparado a bancos de dados relacionais, considerando que as

[4] https://www.mongodb.com/pt-br

pesquisas não trabalham com junções de dados de outras tabelas, mas sim com a incorporação de documentos em um único registro armazenado. Essa estratégia de consulta viabiliza a realização de um agrupamento hierárquico de informações, uma característica diferencial que pode se mostrar útil para aplicações que trabalham com esse tipo de estruturação de dados.

3 ATIVIDADES DESENVOLVIDAS

Neste capítulo, são detalhadas as principais atividades realizadas no estágio, referentes ao desenvolvimento de uma arquitetura de microsserviço focada em soluções de pagamentos. A Seção 3.1 apresenta as características da arquitetura original do sistema. A Seção 3.2 detalha os principais requisitos arquiteturais para o desenvolvimento de uma nova organização arquitetural do sistema. A Seção 3.3 apresenta uma visão geral da nova arquitetura desenvolvida. A Seção 3.4 descreve os principais detalhes sobre como é realizada a comunicação entre os serviços (componentes) do sistema. A Seção 3.5 apresenta os detalhes referentes à implementação do microsserviço de pagamentos. A Seção 3.6 descreve o processo de integração de *gateways* de pagamentos dentro do sistema. Por fim, a Seção 3.7 discute a integração do banco de dados com o microsserviço de pagamentos e detalha alguns desafios técnicos enfrentados durante a realização desse processo.

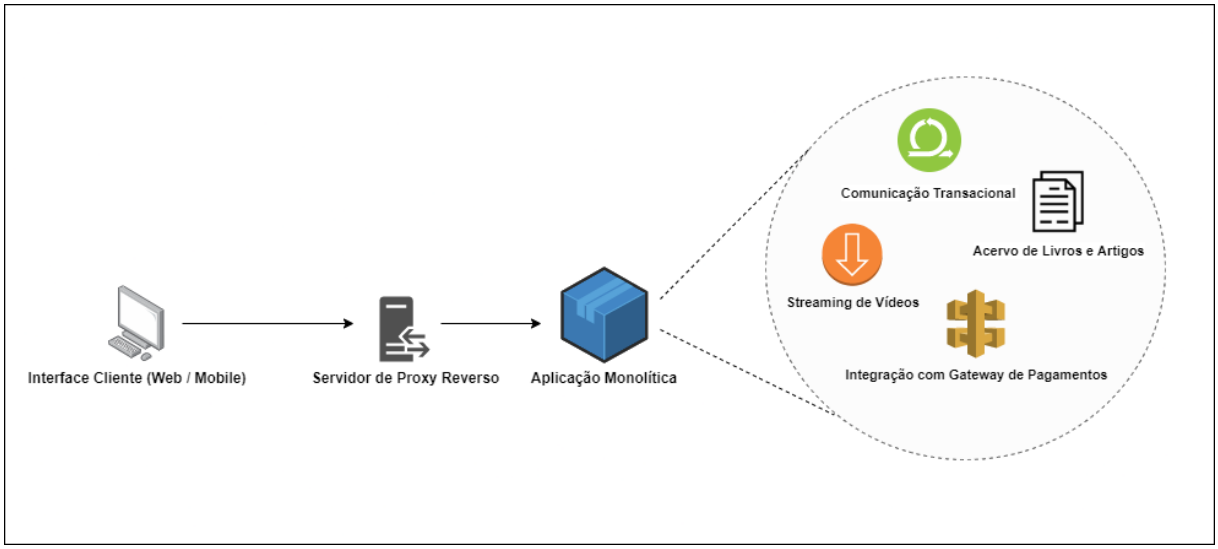
3.1 Arquitetura Original do Sistema

A aplicação Blesss foi desenvolvida utilizando um modelo de arquitetura monolítica e, conforme houve mudanças no modelo de negócio de plataforma, novas funcionalidades também foram adicionadas à estrutura da aplicação. Em vista disso, o processo de construção do serviço inicialmente foi ágil e simples, considerando a facilidade para adicionar e realizar o suporte de todas as funcionalidades do sistema em um único código-fonte. No entanto, esse acúmulo inevitável de responsabilidades resultou em um serviço extremamente complexo, repleto de procedimentos entrelaçados e de difícil manutenção.

A Figura 3.1 apresenta uma demonstração visual de alguns dos principais recursos geridos pela aplicação monolítica. Nesta representação, é possível notar que a aplicação da plataforma Blesss concentra o gerenciamento de todas as funcionalidades necessárias para o modelo de negócio do serviço, que envolvem mecanismos para garantir o(a):

- Distribuição do acervo de publicações textuais (artigos e livros).
- *Streaming* de vídeos e séries.
- Gestão da comunicação transacional com os clientes da plataforma.
- Integração com um *gateway* de pagamentos para processar as transações realizadas pela aplicação.

Figura 3.1 – Antiga organização arquitetural da aplicação Blesss



Fonte: Do autor

A complexidade da aplicação não apenas dificultou o processo de manutenção do código-fonte, como também impôs maiores empecilhos para garantir a escalabilidade e resiliência do sistema como um todo. Portanto, em razão dos diversos problemas técnicos experimentados pela equipe de desenvolvimento no processo de manutenção desse produto, fez-se necessária a criação de um projeto para promover uma reorganização arquitetural da plataforma Blesss, visando reduzir o impacto dessas adversidades e aprimorar a estrutura dos serviços da empresa.

3.2 Principais Requisitos Arquiteturais

Durante a etapa inicial de concepção de um novo modelo arquitetural de serviços, diversas necessidades foram apontadas pela equipe de desenvolvimento do projeto. A junção dessas demandas resultou em uma lista de requisitos técnicos essenciais que a nova arquitetura de microsserviços deveria ser capaz de cumprir. Dentre tais requisitos levantados, destacam-se os seguintes:

- **Baixo Acoplamento:** O sistema deveria ter um baixo acoplamento entre os serviços que compõem sua arquitetura. Isso possibilitaria que as principais aplicações da empresa fossem divididas em componentes menores, tendo em vista a divisão de responsabilidades e a redução de interdependência entre cada serviço.

- **Escalabilidade:** O desacoplamento de serviços deveria tornar o sistema altamente escalável, para possibilitar o aprimoramento e dimensionamento de cada componente da

arquitetura de forma individual. Consequentemente, o sistema poderia ser facilmente ajustado para atender a demanda de seus clientes conforme a carga de trabalho exigida.

- **Facilidade de Manutenção:** Com a divisão das aplicações em componentes menores, o processo de manutenção do sistema tornaria-se mais simples, pois novos recursos ou serviços poderiam ser adicionados sem que todo o sistema fosse alterado. Desse modo, os custos de manutenção e desenvolvimento seriam reduzidos.

- **Reutilização:** Com a separação de responsabilidades entre cada microsserviço da arquitetura, o sistema deveria prover componentes de caráter agnóstico. Ou seja, os serviços criados para determinadas funções poderiam ser reutilizados por outras aplicações da empresa que necessitem desses recursos para realizar suas tarefas.

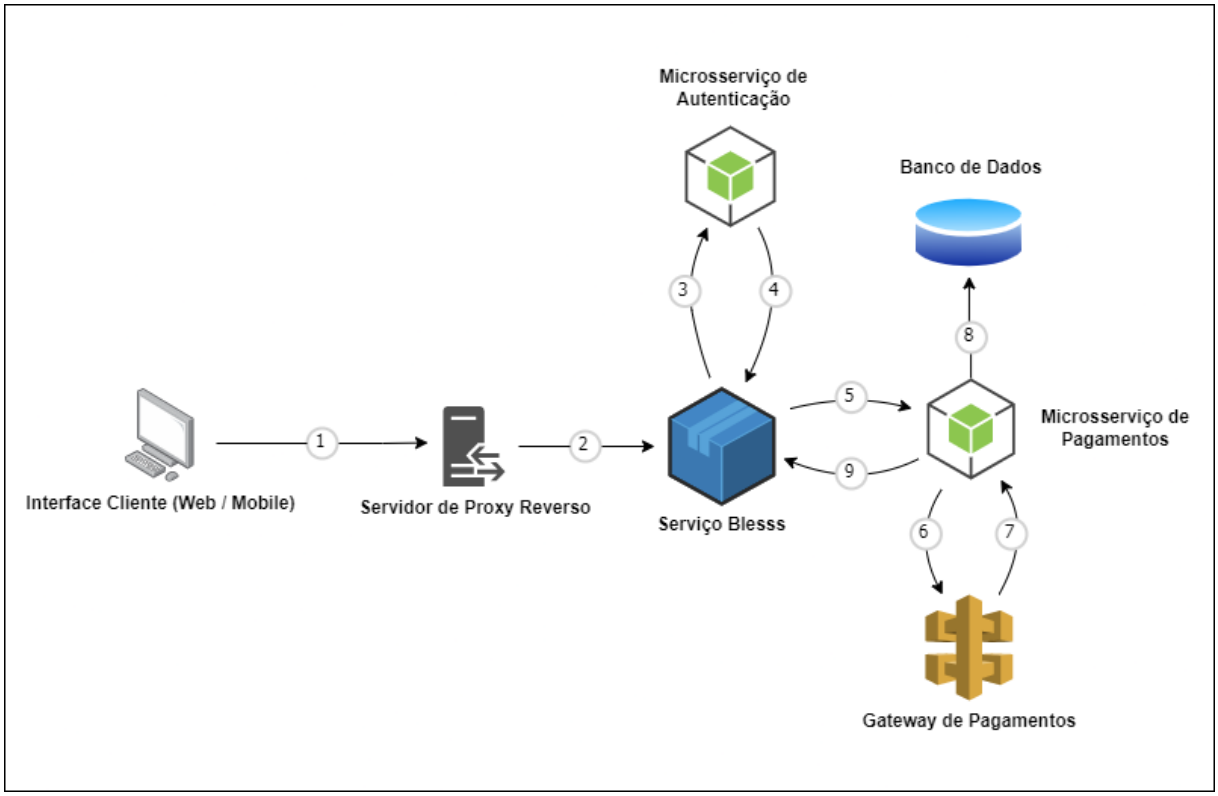
3.3 Visão Geral da Solução Proposta

Visando apaziguar os problemas enfrentados pelo uso de uma arquitetura monolítica, o time técnico responsável pelo projeto trabalhou na criação de um novo sistema, baseado em um modelo de arquitetura de microsserviços. A primeira etapa de desenvolvimento do projeto tinha como objetivo desacoplar o processamento de pagamentos da aplicação principal. A execução desse passo de reestruturação arquitetural permitiria que esse sistema se tornasse mais resiliente e que seus componentes pudessem ser escalados individualmente.

Também esperava-se que, com a implantação da nova organização arquitetural, fosse possível obter determinados benefícios associados à simplificação do processo de manutenção do código-fonte. A flexibilidade para adicionar novos recursos aos microsserviços da arquitetura também era um objetivo importante, pois o modelo de negócio da empresa visava a necessidade de incorporar novas funcionalidades aos serviços da arquitetura, sem que isso viesse a imprimir impactos indesejados sobre os outros componentes da arquitetura final.

A Figura 3.2 apresenta a organização da arquitetura proposta pela equipe, exemplificando o fluxo operacional de um pagamento realizado por um usuário na plataforma Blesss. O fluxo inicia-se com o acesso de um cliente ao website do sistema, com uma tentativa de realizar uma assinatura recorrente para obter acesso aos conteúdos da ferramenta, informando os dados necessários para o processamento da cobrança (1); a solicitação é encaminhada para o respectivo servidor de rede interno, que atua como um *proxy* reverso para encaminhar as requisições aos respectivos serviços requisitados (2).

Figura 3.2 – Nova organização arquitetural dos serviços



Fonte: Do autor

Após a requisição ser redirecionada para a API da plataforma Blesss, esta processará os dados pessoais do usuário e encaminhará uma solicitação de cobrança ao método de pagamento informado pelo cliente; para isso, a aplicação comunica-se com um microsserviço de autenticação para solicitar um *token* de acesso (3). O *token* recebido (4) é utilizado na comunicação com o microsserviço de pagamentos (5), responsável por validar a chave de acesso recebida, e processar a transação financeira descrita. Logo em seguida, o pedido é encaminhado para o gateway de pagamentos (6), que atuará no processamento da transação para retornar uma resposta o resultado da transação (7). Imediatamente após receber a confirmação do pagamento, o microsserviço de pagamentos registra a transação no banco de dados da aplicação (8) e retorna uma resposta à API principal informando a conclusão do processamento (9).

Nesta fase de definição da arquitetura da solução, o envolvimento profissional do estagiário contemplou uma participação colaborativa no processo, com a realização de sugestões pertinentes à concepção do projeto. As etapas de discussão técnica visaram estabelecer definições sobre como o mecanismo de processamento de transações poderia ser desacoplado, como os serviços iriam se comunicar, além de quais recursos e tecnologias poderiam ser utilizadas para construir os novos componentes dessa arquitetura. Por fim, o estagiário também

foi responsável por realizar atividades envolvendo a construção de parte dessa estrutura final, respeitando as restrições e diretrizes de desenvolvimento determinadas pela equipe encarregada.

3.4 Comunicação entre os Serviços

Tendo em vista o modo como os microsserviços da arquitetura foram organizados, também estabeleceu-se a importância de definir um bom design de APIs, que possuísse um esquema de comunicação eficiente entre os serviços interligados. Em razão disso, a equipe de desenvolvimento definiu como uma prioridade técnica o desenvolvimento de aplicações RESTful.

A justificativa para a adoção do estilo arquitetural REST foi embasada na necessidade de implementar uma estrutura de comunicação síncrona e eficaz, baseada em um modelo de requisição/resposta imediata. Desta forma, a utilização desse estilo arquitetural possibilitaria a construção de um sistema simples e confiável, além de facilmente expansível para a integração de novos serviços. Levando isso em consideração, a necessidade de realizar processamentos síncronos de transações financeiras (uma característica importante do modelo de negócio da empresa) foi um importante requisito que pôde ser atendido por meio das vantagens oferecidas pela implementação de uma API RESTful. A plataforma Blesss, por exemplo, trabalha com um retorno imediato de uma confirmação da efetivação do pagamento; no modelo de negócio do produto, o cliente precisa realizar uma assinatura e aguardar pela confirmação do pagamento, para que enfim possa acessar os conteúdos ofertados que são restritos aos usuários assinantes do serviço.

Para viabilizar a comunicação entre os serviços que compõem o sistema, utilizou-se a biblioteca Axios[1] para efetuar requisições HTTP e realizar as devidas manipulações das respostas obtidas conforme as necessidades da aplicação. A Figura 3.3 demonstra um exemplo de uma implementação em JavaScript, utilizando a biblioteca Axios, simulando uma requisição HTTP do tipo POST para o microsserviço de pagamentos para exemplificar o seu funcionamento; este fluxo operacional é uma representação técnica de como exatamente os serviços consumidores (a API da plataforma Blesss, por exemplo) processam as solicitações de transações para o microsserviço de pagamentos, e gerenciam as respostas recebidas.

[1] https://axios-http.com/ptbr/

Figura 3.3 – Exemplo de uma requisição POST implementada por meio da biblioteca axios

```
1  /* Autor: Kaio Vinicius Morais Silva */
2  const axios = require('axios')
3
4  async function createPayment(charge, billing, card) {
5    const body = {
6      charge,
7      billing,
8      card
9    };
10
11   const headers = {
12     'Content-Type': 'application/json',
13     'X-Access-Token': 'authorization_token_example'
14   };
15
16   const uri = 'https://api.payments.blesss.org/payments';
17
18   try {
19     const { data } = await axios.post(
20       uri,
21       body,
22       { headers }
23     );
24
25     return data;
26   } catch (error) {
27     throw new Error('Error processing the requested payment');
28   }
29 }
30
31 module.exports = {
32   createPayment
33 }
```

Fonte: Do autor

A execução do método *createPayment* (linha 4), demonstrado pela Figura 3.3, baseia-se no recebimento de três parâmetros: um objeto *charge* que contém os dados relativos à cobrança solicitada, um objeto *billing* contendo os dados referentes ao cliente a ser cobrado, e um outro objeto *card* contendo os dados do cartão de crédito fornecido pelo cliente para a realização do pagamento; os três objetos são unidos para criar um novo objeto, denominado *body* (linha 5) que será encaminhado na requisição. Em seguida, o atributo *headers* também é criado com o intuito de abrigar um conjunto de informações descritivas a serem associadas ao cabeçalho da requisição HTTP (linha 11); no exemplo demonstrado, são adicionados os campos *Content-Type* e *X-Access-Token*, que definem o tipo do conteúdo transmitido e a chave de autorização a

ser embutida na requisição, respectivamente. Após isso, também é definido um valor constante denominado *uri* (linha 16), que representa o identificador do recurso alvo da requisição.

Posteriormente, para processar a requisição do tipo POST, é realizada uma chamada a função *post* da biblioteca Axios (linha 19), fornecendo os valores de *uri*, *body* e *headers* para o processamento da chamada HTTP. Em razão do método *post* ser uma função assíncrona, a expressão *await* é utilizada para garantir que o processo aguarde pelo retorno de uma *promise* (LORING; MARRON; LEIJEN, 2017) contendo o resultado recebido em reposta à requisição (linha 19). Por estar englobada em uma estrutura *Try-Catch* (da linha 18 até a linha 29), caso a requisição seja bem sucedida, o método *createPayment* retornará uma resposta contendo o resultado da transação e o seu status final de processamento (linha 25); senão, será lançado um erro contendo a descrição do problema ocorrido (linha 27).

A utilização do Axios trouxe inúmeros benefícios que simplificaram o processo de desenvolvimento do canal de comunicações entre as APIs do sistema, considerando algumas vantagens técnicas como:

- Conversão automática de parâmetros e respostas de requisições HTTP em formato JSON para objetos ou outros tipos de dados nativos do JavaScript (AXIOS, 2022).
- Inclusão de recursos de proteção contra alguns tipos de vulnerabilidades de segurança como *Cross Site Forgery*[2].
- Tratamento de erros mais eficiente, por meio do fornecimento de respostas HTTP com informações detalhadas. Tais dados disponibilizados possibilitam a realização de análises mais aprofundadas, visando identificar quaisquer problemas associados às requisições processadas.

3.5 Projeto e Implementação do Microsserviço de Pagamentos

O desenvolvimento do microsserviço de pagamentos representou a primeira tarefa do estagiário no processo de construção do sistema proposto, na qual foram postulados os primeiros requisitos técnicos que tal componente deveria ter perante os requisitos estabelecidos no projeto. A aplicação foi desenvolvida por meio da utilização da tecnologia Node.js, tendo em vista as vantagens oferecidas pela mesma na criação de camadas de serviços com altos ní-

[2] https://owasp.org/www-community/attacks/csrf

veis de escalabilidade e flexibilidade, além de sua exigência relativamente baixa por recursos computacionais (KINSTA, 2021).

O processo de construção do microsserviço de pagamentos também envolveu a utilização do *framework* Express.js[3]. A utilização dessa ferramenta possibilitou que a aplicação final contemplasse um pacote de funcionalidades focadas no desenvolvimento de serviços web, permitindo que o componente assimilasse as características de uma API RESTful.

A implementação desse microsserviço obedeceu uma organização lógica interna do código-fonte, composta por três módulos principais que mapeiam todo o processamento de transações seguindo um fluxo de ações predeterminadas. Os respectivos componentes internos possuem um esquema de interação bem definido, que acatam uma ordem cronológica de passos a serem executados, conforme demonstrado pela Figura 3.4. Tais itens constituintes podem ser descritos como:

- **Módulo de processamento lógico de transações:** Este componente é constituído pelo código-fonte da aplicação responsável por receber e processar o conjunto de dados referentes à solicitação de uma transação financeira.

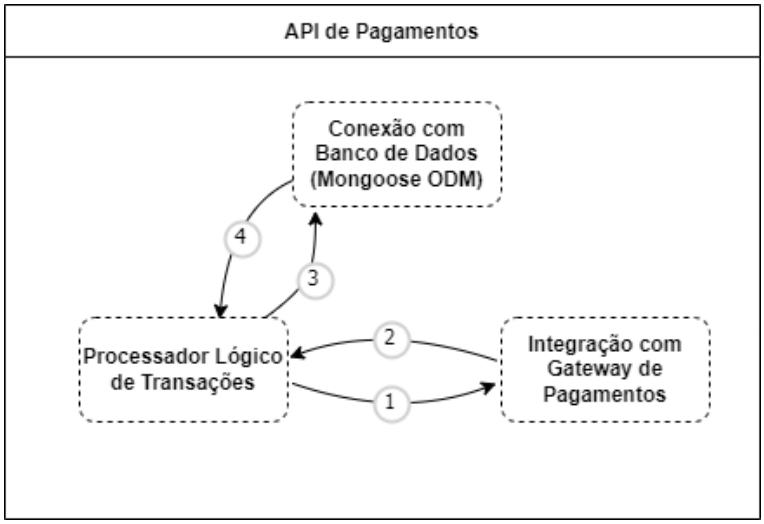
- **Módulo de integração de *gateways* de pagamento:** Este componente contém todo o código-fonte referente à integração de um ou mais *gateways* de pagamentos; por consequência, a partir da agregação das funcionalidades essenciais de tais *gateways*, torna-se possível processar uma transação financeira.

- **Módulo de conexão com o banco de dados:** Este módulo contempla os trechos do código-fonte em que é processada a conexão com o banco de dados, além da disponibilização de funcionalidades que viabilizam os processos de leitura e escrita de dados.

O fluxo de dados, conforme apresentado pela Figura 3.4, inicia-se a partir do recebimento dos dados referentes à transação financeira solicitada pelo módulo de processamento lógico, etapa na qual é validado o formato e a integridade de tais informações. Após a validação, a solicitação de pagamento é repassada ao respectivo *gateway* de pagamentos integrado (1). Com o recebimento do pedido, o *gateway* de pagamentos responsabiliza-se por encaminhar tal solicitação à adquirente responsável por analisar a integridade da transação e validar a possibilidade da realização do pagamento com o respectivo banco emissor; ao final do processamento, o

[3] https://expressjs.com/pt-br/

Figura 3.4 – Organização Interna do Microsserviço de Pagamentos



Fonte: Do autor

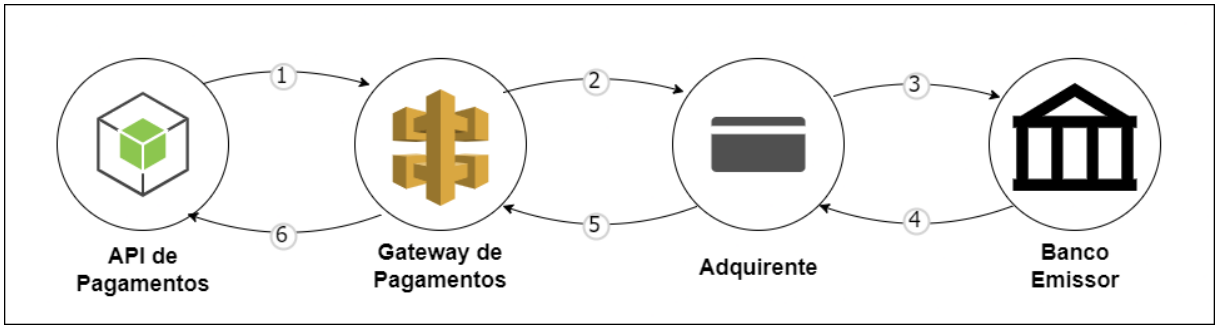
respectivo *gateway* retorna uma resposta definitiva para o componente de processamento lógico, informando se a transação foi bem sucedida ou não (2). Após receber o resultado da transação processada pelo *gateway* de pagamentos, a camada de processamento da API manipula as informações necessárias, e solicita para o banco de dados da aplicação a criação de um registro do pagamento realizado (3). Logo em seguida, o componente de conexão com o banco de dados retorna uma resposta para confirmar a criação do registro (4). Por fim, o microsserviço envia uma resposta final para a aplicação consumidora, contendo as informações referentes ao status de processamento do pagamento solicitado.

3.6 Integração com um *Gateway* de Pagamentos

Buscando viabilizar o processamento de transações, a equipe responsável pelo projeto também designou ao estagiário a tarefa de realizar a integração do microsserviço de pagamentos com um *gateway* de pagamentos. Essas ferramentas são responsáveis por garantir a comunicação entre um cliente e um banco emissor, por meio de empresas adquirentes, para possibilitar a realização de pagamentos digitais (PAGAR, 2020). Por consequência, todas as etapas de processamento do pagamento são delegadas ao respectivo sistema externo, removendo a responsabilidade do serviço consumidor em processar as transações solicitadas.

A Figura 3.5 apresenta uma demonstração visual da integração do microsserviço de pagamentos com o *gateway* especializado, contendo os principais passos realizados durante o processamento de uma transação de cartão de crédito. O fluxo de operações inicia-se com o

Figura 3.5 – Integração com gateway de pagamentos



Fonte: Do autor

microsserviço de pagamentos transmitindo uma solicitação de pagamento para o respectivo *gateway* integrado. Essa solicitação contém os dados do cliente, a descrição da cobrança realizada e do método de pagamento selecionado (1). Após o recebimento da solicitação, o *gateway* de pagamentos encaminha os dados da cobrança para a adquirente associada, que é responsável por analisar a integridade das informações da transação (2). O processo de validação da cobrança, no entanto, envolve uma comunicação entre a adquirente e o banco emissor, na qual a respectiva instituição financeira responsabiliza-se por verificar a segurança dos dados recebidos e conferir se o cliente final informado possui um saldo suficiente para confirmar o resgate do valor; se sim, a transação é efetivada (3). Posteriormente, o banco emissor emite uma resposta para a adquirente, com os dados gerados pelo processamento da cobrança (4); a adquirente, por sua vez, encaminha os resultados para o *gateway* de pagamentos (5). Por fim, o *gateway* retorna uma resposta final para o microsserviço de pagamentos, com a confirmação da transação e as informações geradas pelo seu processamento (6).

A utilização de um *gateway* de pagamentos dentro dessa nova estrutura proveu uma redução dos custos de desenvolvimento, por viabilizar uma integração entre o microsserviço de pagamentos com as instituições financeiras de forma simples e direta. A garantia de um maior nível de segurança sobre o processamento das transações também apresentou-se como uma justificativa plausível para a adoção dessa ferramenta, considerando que a mesma atua em conformidade com as normas estabelecidas pelo PCI DSS (COUNCIL, 2022) para transmitir e armazenar os dados das transações de forma segura. Esse sistema também apresenta mecanismos antifraude, responsáveis por analisar a integridade de um pagamento e estimar um risco associado ao seu processamento. A estimativa de risco gerada indica a possibilidade de aprovação ou rejeição da cobrança antes mesmo de encaminhar tal requisição para a adquirente

responsável. Esse recurso também foi um fator importante que contribuiu para que a empresa optasse pela contratação desse serviço.

3.7 Integração com o Banco de Dados

Com o objetivo de registrar todas as transações processadas pelo microsserviço de pagamentos, o sistema final contemplou a integração de um banco de dados não relacional (NoSQL) orientado a documentos, sendo essa mais uma das atividades alocadas ao estagiário. Para isso, priorizou-se a adoção do MongoDB como a alternativa mais adequada para armazenar todo o volume de dados gerado a partir dos pagamentos processados pela aplicação. A justificativa pela adoção de uma base de dados fundamentada em um modelo não relacional baseia-se nas vantagens identificadas juntamente com a equipe responsável pelo projeto, entre as quais pode-se destacar:

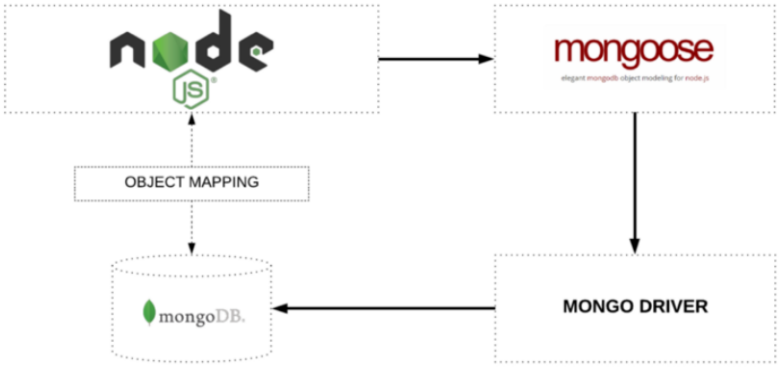
- Flexibilidade para elaborar esquemas de dados simplificados, tornando o processo de desenvolvimento mais ágil e simples.
- Maior nível de escalabilidade, a partir da implantação da base de dados em clusters distribuídos para que seja possível garantir sua disponibilidade.
- Alta disponibilidade de recursos e bibliotecas, compatíveis com o Node.js, que auxiliam no processo de desenvolvimento e integração do banco de dados MongoDB.

O processo de integração do microsserviço de pagamentos com o MongoDB contou com a utilização da biblioteca Mongoose[4] como um mapeador entre documentos e objetos (ODM), responsável por processar as informações enviadas pelo serviço e armazenar os dados em registros (documentos) do tipo JSON, como demonstrado pela Figura 3.6. Com a adoção de tal recurso, os processos de mapeamento, registro e consulta de informações foram simplificados. Isso contribuiu para agilizar o desenvolvimento da aplicação ao delegar as responsabilidades citadas para a biblioteca, que passou a atuar como uma espécie de intermediador entre o banco de dados MongoDB com o serviço desenvolvido em Node.js.

Obedecendo à regra de negócio do microsserviço de pagamentos, todas as transações financeiras são registradas na base de dados do serviço. Para isso, o banco de dados armazena

[4] https://mongoosejs.com/

Figura 3.6 – Mapeamento de objetos entre o Node.js e MongoDB gerenciado pelo Mongoose



Fonte: (KARNIK, 2018)

dois tipos de coleções de dados, denominadas *transactions* e *subscriptions*. A coleção *transactions* é responsável por armazenar transações de pagamentos avulsas, na qual cada documento registrado contém as seguintes informações:

- Método de pagamento (cartão de crédito ou boleto bancário).
- Valor da cobrança.
- Data de processamento.
- Status de processamento pelo *gateway* de pagamentos (aguardando por pagamento, pago ou recusado).
- Descrição do tipo de produto adquirido ou serviço contratado em função do pagamento realizado.

Por outro lado, a coleção *subscriptions* responsabiliza-se por armazenar todas as transações referentes a registros de assinaturas recorrentes. Esse modelo de pagamento apenas aceita cartões de crédito como um método de pagamento válido e trabalha com a emissão recorrente de cobranças, justificadas pela prestação de um determinado serviço ao cliente final. Cada registro dessa coleção possui os seguintes atributos:

- Modelo de recorrência do pagamento (mensal ou anual).
- Valor da cobrança.
- Lista de transações prévias, contendo as datas e status de processamento de cada cobrança realizada.

- Data do próximo pagamento a ser realizado.
- Status atual da assinatura (pago, cancelado ou pagamento pendente).
- Descrição do tipo de serviço contratado em função do pagamento realizado.

3.7.1 Desafios Encontrados

Apesar das coleções de dados descritas apresentarem características distintas referentes aos tipos de pagamentos existentes, ambas também incluem um atributo de identificação única relativa ao cliente cobrado pelo sistema. Esse identificador representa um tipo de chave estrangeira (MANNINO, 2008), responsável por referenciar um cadastro desse consumidor pertencente a um outro banco de dados. Essa base de dados é administrada por um outro microsserviço interno da empresa (não abordado na arquitetura descrita neste projeto), cuja especialidade é focada no gerenciamento de usuários cadastrados em todas as aplicações da empresa; a base de usuários da plataforma Blesss, por exemplo, é gerenciada por esse serviço.

A utilização de uma chave estrangeira para referenciar registros externos de usuários nos documentos de transações foi um passo bastante discutido pela equipe de desenvolvimento. A implementação dessa estratégia pode ser considerada uma controvérsia arquitetural, por ferir o princípio de integridade referencial de bancos de dados (BLAHA, 2005). No cenário descrito, se o sistema não tomar as devidas precauções, as transações registradas pelo microsserviço de pagamentos poderiam sofrer com inconsistência de dados, caso fossem excluídos os cadastros de usuários que estivessem sendo referenciados por tais transações. Naturalmente, a utilização de um único banco relacional poderia ser uma alternativa para resolver esse problema, o que resultaria em uma centralização de todos os registros em uma única base de dados, com o estabelecimento de relacionamentos entre as tabelas de dados para garantir a consistência das informações armazenadas (MANNINO, 2008).

Entretanto, a existência de um microsserviço especializado na gestão de usuários era um importante requisito do modelo de negócios empregado pela empresa. Portanto, ciente das controvérsias técnicas e visando apaziguar possíveis complicações futuras, a equipe trabalhou em conjunto para implementar restrições de negócio no serviço de gerenciamento de usuários. Tais medidas restritivas prezavam pela garantia de que nenhum registro de um cliente fosse removido ou atualizado de modo inadequado. Ou seja, as aplicações consumidoras desse serviço (incluindo a aplicação Blesss) não seriam capazes de excluir cadastros de usuários de forma

indevida. Consequentemente, o controle de consistência de dados de usuários tornou-se uma responsabilidade do microsserviço de gestão dos usuários (clientes) das aplicações desenvolvidas pela empresa.

4 CONCLUSÃO

A implantação da nova arquitetura de serviços dentro da empresa Zeester trouxe resultados positivos para a plataforma Blesss, pois ela permitiu que o sistema de processamento de pagamentos fosse desacoplado da aplicação para se tornar um microsserviço especializado. Em razão disso, a nova arquitetura adotada permitiu que esse sistema de processamento de transações pudesse ser mantido, aprimorado e escalado com maior eficiência, sem que isso impactasse o funcionamento dos serviços oferecidos pelo Blesss. Com o desacoplamento desses serviços, juntamente com a criação de um microsserviço de pagamentos com um código-fonte melhor desenvolvido, a empresa ganhou flexibilidade para escolher *gateways* de pagamentos que se adequem ao seu modelo de negócio, o que também foi um benefício muito importante. A reorganização arquitetural também viabilizou a reutilização do mesmo sistema de pagamentos em um outro produto interno que veio a ser desenvolvido posteriormente pela Zeester, no segundo semestre de 2021, para prover uma outra solução digital para um cliente da empresa. Por conseguinte, isso implicou na redução de custos e tempo gasto no desenvolvimento da nova aplicação.

No entanto, ainda existem melhorias que poderiam ser implementadas na arquitetura de serviços desenvolvida, a fim de trazer outros tipos de benefícios para os produtos da empresa no futuro. Uma possível melhoria seria a adição de um mecanismo de processamento assíncrono de transações, que implicaria no enfileiramento das solicitações de cobranças financeiras, por parte do microsserviço de pagamentos. Essa modificação visaria encaminhar tais requisições aos *gateways* de pagamentos, para assim receber as respectivas respostas de forma assíncrona. A adição desse recurso seria uma alternativa interessante para futuras aplicações da empresa que almejam processar pagamentos, mas de forma escalável, sem necessariamente prover respostas síncronas que podem comprometer o tempo de resposta do sistema às ações dos clientes finais (caso isso seja apresentado como um requisito de modelo de negócio).

Por fim, a realização do estágio representou uma grande oportunidade de aprendizado, pois essa experiência permitiu a aplicação de diversos conhecimentos obtidos durante a graduação em Ciência da Computação em um contexto profissional. Em função disso, disciplinas como Programação Web, Sistemas Operacionais, Estrutura de Dados, Engenharia de Software, Redes de Computadores e Introdução a Banco de Dados foram bases de conhecimento essenciais para a realização das atividades executadas no estágio. No entanto, o curso também poderia trazer disciplinas com um foco maior no estudo de modelagem de arquiteturas de sistemas,

visando oferecer para os alunos uma importante base de conhecimento técnico que possa ser aproveitada em um contexto de atuação profissional no mercado.

Em contrapartida, as tarefas realizadas também representaram uma experiência muito enriquecedora para a formação profissional. A participação no projeto de concepção da arquitetura de serviços foi uma experiência inigualável para o estagiário, considerando todos os conceitos e tecnologias aprendidas, além de todos os desafios técnicos enfrentados durante o processo de desenvolvimento da infraestrutura do sistema. O trabalho em equipe gerenciado por uma metodologia ágil e a assimilação de tarefas para resolver problemas em um contexto de atuação profissional, por exemplo, também foram etapas cruciais para a criação de uma base de conhecimento sólida. Por fim, a carga de experiências gerada foi essencial para a construção da carreira profissional do estagiário, permitindo que ele pudesse ingressar no mercado de trabalho em busca de novas oportunidades profissionais.

REFERÊNCIAS

ADSERVIO. **Event-Driven Architecture**. 2022. Disponível em: <https://www.adservio.fr/post/event-driven-architecture>. Acesso em: 17 jul. 2022.

AGRE, P. E. P2p and the promise of internet. **COMMUNICATIONS OF THE ACM**, v. 46, n. 2, p. 39, 2003.

ATLASSIAN. **Microservices vs. monolithic architecture**. 2022. Disponível em: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>. Acesso em: 26 jun. 2022.

AXIOS. **Introdução**. 2022. Disponível em: <https://axios-http.com/ptbr/docs/intro>. Acesso em: 24 jul. 2022.

BANGARE, S. et al. Using node. js to build high speed and scalable backend database server. In: **Proc. NCPCI. Conf**. [S.l.: s.n.], 2016. v. 2016, p. 19.

BANKER, K. et al. **MongoDB in action: covers MongoDB version 3.0**. [S.l.]: Simon and Schuster, 2016.

BIEHL, M. **RESTful Api Design**. [S.l.]: API-University Press, 2016. v. 3.

BLAHA, M. Referential integrity is important for databases. **Modelsoft Consulting Corp**, 2005.

COUNCIL, P. S. S. **PCI Security**. 2022. Disponível em: <https://pt.pcisecuritystandards.org/pci_security/>. Acesso em: 24 jul. 2022.

ETZION, O. Towards an event-driven architecture: An infrastructure for event processing position paper. In: SPRINGER. **International Workshop on Rules and Rule Markup Languages for the Semantic Web**. [S.l.], 2005. p. 1–7.

FOWLER, J. L. M. **Microservices: a definition of this new architectural term**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 26 jun. 2022.

GEEKSFORGEEKS. **Node.js Event Loop**. 2021. Disponível em: <https://www.geeksforgeeks.org/node-js-event-loop/>. Acesso em: 13 ago. 2022.

KARNIK, N. **Introduction to Mongoose for MongoDB**. 2018. Disponível em: <https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/>. Acesso em: 24 jul. 2022.

KINSTA. **O Que é Node.js e Por Que Usá-lo?** 2021. Disponível em: <https://kinsta.com/pt/base-de-conhecimento/o-que-e-node-js/>. Acesso em: 24 jul. 2022.

LORING, M. C.; MARRON, M.; LEIJEN, D. Semantics of asynchronous javascript. In: **Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages**. [S.l.: s.n.], 2017. p. 51–62.

LUCIO, J. P. D. et al. Análise comparativa entre arquitetura monolítica e de microsserviços. Florianópolis, SC, 2017.

MANNINO, M. V. **Projeto, Desenvolvimento de Aplicações e Administração de Banco de Dados-3**. [S.l.]: AMGH Editora, 2008.

MENARD, N. Decision criteria between microservice and monolithic architecture. 2020.

MICHELSON, B. M. Event-driven architecture overview. **Patricia Seybold Group**, v. 2, n. 12, p. 10–1571, 2006.

MICROSOFT. **API Gateways**. 2022. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/microservices/design/gateway>. Acesso em: 20 ago. 2022.

MONGODB. **BSON Types**. 2022. Disponível em: <https://www.mongodb.com/docs/manual/reference/bson-types/>. Acesso em: 19 jun. 2022.

NASCIMENTO, M.; UNIFACCAMP, C. **MongoDB: Um Estudo Teórico-Prático do Conceito de Banco de Dados NoSQL - Trabalho de Diplomação**. Tese (Doutorado), 12 2014.

OLIVEIRA, C.; ZANETTI, H. **Node.js: programe de forma rápida e prática**. Saraiva Educação S.A., 2021. ISBN 9786558110217. Disponível em: <https://books.google.com.br/books?id=QnRIEAAAQBAJ>.

OLUWATOSIN, H. S. Client-server model. **IOSR Journal of Computer Engineering**, v. 16, n. 1, p. 67–71, 2014.

PAGAR. **Tudo o que você precisa saber sobre bandeira de cartão, gateway, adquirente e subadquirente**. 2020. Disponível em: <https://pagar.me/blog/subadquirente-e-adquirente-o-que-voce-precisa-saber/>. Acesso em: 24 jul. 2022.

PEREIRA, C. R. **Aplicações web real-time com Node. js**. [S.l.]: Editora Casa do Código, 2014.

RAMOS, F. M. P. et al. Definição de uma arquitetura p2p baseada em reputação e orientada a serviços. Universidade Federal do Maranhão, 2009.

REDHAT. **O que é uma arquitetura orientada por eventos?** 2019. Disponível em: <https://www.redhat.com/pt-br/topics/integration/what-is-event-driven-architecture>. Acesso em: 02 jul. 2022.

REDHAT. **What is a REST API**. 2020. Disponível em: <https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api>. Acesso em: 19 jun. 2022.

RIGHI, R. R.; PELLISSARI, F. R.; WESTPHALL, C. M. Escambo: um modelo de reputação e micropagamentos para sistemas peer-to-peer. In: **IV Congresso Brasileiro de Computação-CBComp**. [S.l.: s.n.], 2004.

SERVICES, A. W. **What is an Event-Driven Architecture?** 2022. Disponível em: <https://aws.amazon.com/pt/event-driven-architecture/>. Acesso em: 02 jul. 2022.

SOARES, M. dos S. Metodologias ágeis extreme programming e scrum para o desenvolvimento de software. **Revista Eletrônica de Sistemas de Informação**, v. 3, n. 1, 2004.

SOFTWARE, O. **Micro Serviços: Qual a diferença para a Arquitetura Monolítica?** 2021. Disponível em: <https://www.opus-software.com.br/micro-servicos-arquietura-monolitica/>. Acesso em: 26 jun. 2022.

SOMMERLAD, P. Reverse proxy patterns. In: **EuroPLoP**. [S.l.: s.n.], 2003. p. 431–458.

TAPIA, F. et al. From monolithic systems to microservices: A comparative study of performance. **Applied sciences**, MDPI, v. 10, n. 17, p. 5797, 2020.

TSAI, W.-T. et al. Scenario-based functional regression testing. In: IEEE. **25th Annual International Computer Software and Applications Conference. COMPSAC 2001**. [S.l.], 2001. p. 496–501.