Arquitetura de Microsserviços Back-end em uma Aplicação de Pequena Escala: Estudo de Caso Escola Aconchego

Luan Rodrigues de Oliveira[1], Pablo Freire Matos[2]

[1]Discente Superior em BSI, [2]Docente de Informática

Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA)
Av. Sérgio Vieira de Mello, 3.150, Zabelê, 45.078-900 - Vitória da Conquista - BA - Brasil

{luanrodrigues51, pablofmatos}@gmail.com

Abstract. *This paper presents a case study for a small-scale microservices based back-end application whose domain is a dance school in Vitória da Conquista, BA. The work consists of gathering application requirements, followed by developing the necessary resources. As a result of this work, a qualitative analysis was made from the developer's point of view, highlighting the negative and positive points of using microservices in a small-scale application.*

Resumo. *Esse artigo apresenta um estudo de caso para uma aplicação back-end em microsserviços de pequena escala que tem como domínio uma escola de dança em Vitória da Conquista, BA. O trabalho consiste no levantamento de requisitos da aplicação, seguida pelo desenvolvimento dos recursos necessários. Com resultado deste trabalho, foi feita uma análise qualitativa do ponto de vista do desenvolvedor, destacando os pontos negativos e positivos do uso de microsserviços em uma aplicação de pequena escala.*

1. Introdução

Segundo Söylemez, Tekinerdogan e Tarhan (2022), os sistemas distribuídos ganharam destaque na última década com a crescente demanda por Internet das Coisas e avanços na computação em nuvem. Esses sistemas vêm sofrendo mudanças na forma como são organizados nos últimos 60 anos devido às novas demandas tecnológicas e à evolução na infraestrutura. Tais mudanças e a crescente complexidade dos sistemas geraram a necessidade de criar novos padrões de organização.

Como afirma Lindsay *et al*. (2021), os grandes *mainframes* do início da década de 60 proviam acesso através de terminais para que vários clientes locais pudessem realizar operações simultaneamente. Tal organização é chamada de arquitetura cliente-servidor, que hoje é a forma padrão de comunicação da web. Após isso, ao final da década, de acordo com os autores, surgiu a técnica de comutação de pacotes, que permitia enviar uma mensagem de dados dividida em pequenas unidades. Essa técnica possibilita que os pacotes sejam enviados sem uma ordem definida e por vários caminhos diferentes. Através dela foi concebível a criação de *clusters* de computadores, que permitiam combinar o desempenho de diversos microprocessadores, que eram conectados em rede, gerando uma alternativa aos poderosos, porém caros, supercomputadores.

Os padrões de sistemas distribuídos continuaram evoluindo, passando pelo surgimento dos computadores pessoais até a criação da *World Wide Web*. Chegando ao século 21, a explosão no desenvolvimento de serviços web levou à criação da arquitetura orientada a serviços. O objetivo dessa arquitetura é melhor adequar os sistemas aos requisitos da computação distribuída. Nela, recursos de *software* são vistos como serviços, contendo em si módulos bem definidos e interfaces nítidas que provêm as funcionalidades necessárias sem precisar de outros pacotes [Papazoglou e Van Den

Heuvel 2007].

Ao evoluir a partir da arquitetura orientada a serviços e ao buscar solucionar alguns de seus problemas, como disponibilidade, tolerância a falhas e escalabilidade, surge, na década de 2010, a arquitetura de microsserviços [Söylemez, Tekinerdogan e Tarhan 2022]. As aplicações práticas de microsserviços que obtiveram maior alcance foram experimentos feitos em empresas de grande escala como Netflix, Amazon ou Spotify [Bogner *et al.* 2019]. Similarmente à arquitetura orientada a serviços, microsserviços constituem uma abordagem de modularização, onde cada serviço deve ser criado tendo em vista um conjunto de funcionalidades que podem ser agrupadas. No entanto, os microsserviços dão ainda mais ênfase à independência entre os serviços, com processos de publicação distintos e controle descentralizado [Jamshidi *et al.* 2018]. Esses serviços se comunicam através de uma rede, expondo suas funcionalidades através de interfaces. Eles também encapsulam a forma de lidar com bancos de dados, com cada serviço cuidando da recuperação ou salvamento dos dados conforme necessário, sem expor o banco completamente [Newman 2021].

Atualmente, os microsserviços se tornaram altamente populares. A pesquisa, feita pela comunidade online Dzone e publicada por Glen (2018), apresenta que aproximadamente 50% dos desenvolvedores estão utilizando microsserviços, enquanto 38,7% estão considerando o uso (Figura 1). Ainda segundo a pesquisa, os motivos mais citados para uso foram escalabilidade e publicação rápida, seguido pela habilidade de trabalhar com times mais independentes e focados em um único produto. Entre a fatia dos usuários que não estavam utilizando e nem consideravam utilizar microsserviços, o que corresponde a 11,6% do total, 75% afirmaram que o motivo era a falta de conhecimento ou a ausência de casos aplicáveis.

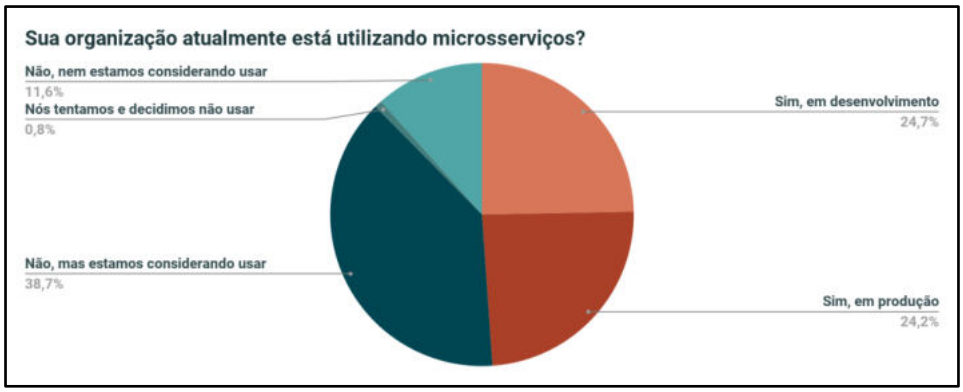

Figura 1. Adoção de microsserviços. Fonte: Glen (2018)

Dados semelhantes também podem ser encontrados em um levantamento feito por Knoche e Hasselbring (2019), que coletaram dados sobre microsserviços em empresas alemãs. Segundo as informações levantadas, os maiores motivadores para implementação de microsserviços são alta escalabilidade, elasticidade e disponibilidade, além da possibilidade de colocar o produto mais rápido no mercado. Enquanto isso, as maiores barreiras são conhecimento técnico insuficiente, complexidade e problemas de compatibilidade.

Em relação ao sucesso da implementação de microsserviços, na Figura 2 é demonstrada a avaliação de sucesso em organizações em relação à implementação de microsserviços publicada por Loukides e Swoyer (2020). É possível notar que a maior parte das organizações que implementaram microsserviços se sentem satisfeitas com os

benefícios obtidos.

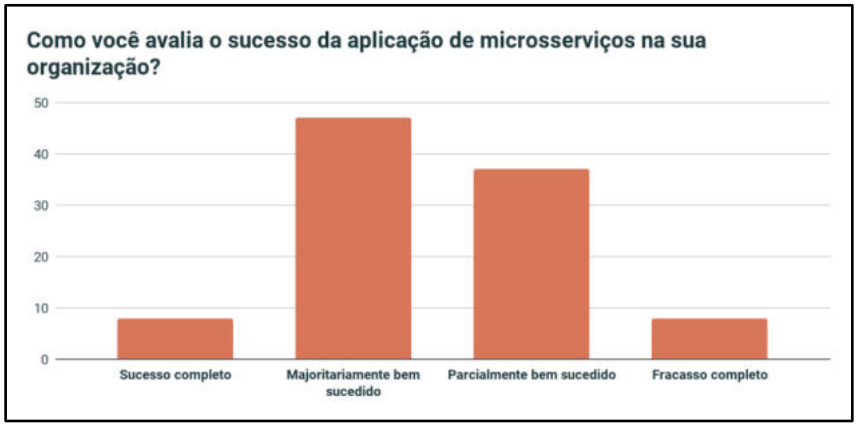

Figura 2. Avaliação de sucesso dos microsserviços. Fonte: Loukides e Swoyer (2020)

A pesquisa divulgada por Glen (2018) também demonstra que 80% dos desenvolvedores afirmaram que usar microsserviços deixou seu trabalho mais fácil. Por consequência desses resultados e do fácil acesso às ferramentas de código aberto que ajudam a implementar esse padrão, muitas empresas estão considerando os microsserviços como uma opção viável para implementação de seus sistemas.

No entanto, apesar da supracitada popularidade dos microsserviços, ainda existem poucos estudos acadêmicos na área [Bogner *et al.* 2019]. A falta se dá principalmente de estudos de caso que demonstrem a implementação em um domínio de pequena escala. Essa é uma questão relevante, já que uma das principais críticas à arquitetura de microsserviços é que ela é mais adequada para aplicações de grande escala, com muitos serviços e equipes trabalhando em conjunto. No entanto, há uma crescente demanda por soluções de *software* mais ágeis e escaláveis, mesmo em aplicações menores. Isso levanta a questão sobre se a arquitetura de microsserviços é viável para aplicações de pequena escala ou se ela é excessivamente complexa e custosa para esses casos. Newman (2021) argumenta que a arquitetura de microsserviços pode ser adaptada para aplicações menores, desde que seja feita uma implementação cuidadosa e estratégica. No entanto, Richardson (2018) afirma que a arquitetura monolítica é mais adequada para aplicações menores, já que é mais simples e fácil de gerenciar.

Este trabalho tem como objetivo investigar essa questão e realizar um estudo de caso envolvendo o desenvolvimento de uma aplicação *back-end* em microsserviços utilizando um domínio de pequena escala. A opção de se utilizar de um estudo de caso como metodologia foi feito, pois, segundo Wazlawick (2011), se trata de um método útil para testar um conceito em uma situação pouco documentada. O estudo também pode ser classificado como exploratório, pois será feita uma análise qualitativa dos resultados, levando em consideração as experiências de desenvolvimento. O projeto escolhido é definido como de pequena escala, pois, segundo determinações do artigo publicado pela Oregon State University (2014), suas regras de negócio envolvem pouco risco e complexidade e existe apenas um dono do produto.

O resultado deste trabalho pode contribuir para a tomada de decisão sobre a escolha da arquitetura mais adequada para diferentes tipos de aplicações de *software*. O artigo organiza-se da seguinte forma: na Seção 2, serão tratados os fundamentos que baseiam o trabalho; na Seção 3, haverá uma discussão sobre os trabalhos correlatos; na Seção 4, será demonstrado o processo de desenvolvimento da aplicação. Por fim, na

Seção 5, serão destacadas as considerações finais.

2. Referencial Teórico

Nesta seção serão tratados os conceitos que fundamentam o trabalho.

2.1 Microsserviços

Segundo Newman (2021), microsserviços são serviços independentes que são modelados tendo em mente um domínio de negócio. Podem existir serviços para quaisquer propósitos. Um pode ficar responsável pelo pagamento e outro pelo gerenciamento de usuários, por exemplo. Juntos, eles constituem um ecossistema que pode formar um ou diversos produtos. Um microsserviço pode ser montado como uma caixa preta, tendo somente suas interfaces disponíveis para uso. Isso é particularmente útil quando se quer abstrair regras de negócio complexas, já que não é preciso se preocupar com a implementação de um certo trecho do código. O consumidor do serviço poderia assim então cuidar apenas de suas próprias funcionalidades.

Serão tratadas nesta subseção os principais pontos característicos na utilização de microsserviços. Em um ambiente de produção de *software*, assim que uma aplicação está pronta para ser utilizada pelos consumidores, é feito o processo de publicação ou *deploy*. Esse processo consiste em diversas etapas e visa deixar a aplicação disponível para uso externo [Nygard 2012]. Os microsserviços podem facilitar o processo de publicação de novas funcionalidades, pois a aplicação estaria dividida em diferentes partes, mais simples de gerenciar. Porém, para que seja efetivo, o processo de publicação de um microsserviço não deve depender de outros serviços para que uma nova versão seja publicada. É preciso, então, definir de forma correta cada escopo, para que o funcionamento de uma lógica de negócio não esteja acoplada entre diversos microsserviços. Além disso, é preciso deixar bem documentado como acontecerão as comunicações entre os serviços. Do contrário, uma publicação pode gerar comportamentos não esperados [Fowler e Lewis 2014].

Além do processo de publicação, é importante olhar para a capacidade de manutenção de um *software*. Ela dita a quantidade de esforço necessária para manter um sistema. Esse esforço pode ser medido de acordo com diversas métricas. Em empresas, geralmente se refere ao tempo gasto por um ou mais desenvolvedores a fim de aprimorar ou resolver problemas em uma determinada aplicação [Newman 2021]. Segundo Lewis e Fowler (2014), esse custo tende a ser maior no caso de aplicações grandes e com diversas regras de negócio. De acordo com os autores, uma das dificuldades, por exemplo, é de adicionar novos membros ao time, já que é preciso introduzir diversas regras de negócio. Como consequência, um desenvolvedor teria que ser alocado para ensinar como o projeto funciona ou montar uma documentação vasta. Os microsserviços seriam então uma possível solução, já que tendem a focar em soluções mais bem definidas e menores. Eles também facilitam o uso de times ágeis, já que cada time poderia ficar responsável por apenas um ou poucos microsserviços, tendo seu escopo reduzido e sua produtividade aumentada.

Outro ponto bastante abordado quando se trata de microsserviços é a escalabilidade, que, segundo Bondi (2000), é a habilidade de um sistema em continuar funcionando corretamente enquanto seu número de usuários ou requisitos cresce. Esse crescimento saudável envolve não somente a capacidade de suportar a carga de novos consumidores, mas também de ser aberto a futuras mudanças em suas regras de negócio. Como os microsserviços podem ser focados em uma área específica do negócio, é

possível utilizar-se das tecnologias e arquiteturas adequadas. Por exemplo, um sistema que recebe um grande número de requisições por segundo pode optar por um banco que seja compatível com tal configuração. Da mesma forma seria caso existisse algum outro serviço que precisasse lidar com cálculos complexos. Além disso, pela natureza modular dos microsserviços, poderia ser mais fácil diagnosticar e corrigir problemas de desempenho [Newman 2021].

2.2 Comunicação entre Serviços

Os microsserviços podem se comunicar de forma síncrona ou assíncrona. Na forma síncrona, é utilizada a arquitetura *REST*. Já na forma assíncrona, os serviços se comunicam através de eventos. A comunicação via *REST* estabelece uma série de princípios e regras para garantir uma comunicação coerente. Trata-se de uma arquitetura cliente-servidor, sem estado e que se baseia em recursos. Cada recurso da *API* fornecerá uma série de ações a serem feitas em uma determinada entidade [Gupta 2023]. Já na comunicação baseada em eventos, cada serviço poderá consumir ou produzir eventos para os demais de acordo com a necessidade. Cada evento conterá em si as informações necessárias a serem capturadas. Sendo assim, os serviços podem se comunicar indiretamente [Iturralde 2020].

2.3 Tecnologias de Desenvolvimento

Nesta subseção serão discutidas as tecnologias utilizadas no desenvolvimento deste trabalho.

2.3.1 Node.js

De acordo com Tilkov e Vinoski (2010), Node.js é um ambiente Javascript voltado a servidores que se baseia no V8, um interpretador criado pela empresa de tecnologia Google. Ele é focado em desempenho e pouco uso de memória, com excelência em processos com alto índice de entrada e saída de dados. Através dele, é possível executar *scripts* escritos em *Javascript* no servidor.

O Node.js opera segundo a lógica de seu *loop de* eventos. Apesar de operar em uma única *thread*, ele é capaz de gerenciar diversas tarefas ao mesmo tempo. Cada nova tarefa enviada ao interpretador é colocada em uma pilha. Caso se trate de uma tarefa síncrona, com processamento imediato, ela é enviada imediatamente para uma fila de execução. Do contrário, a tarefa é enviada para execução em segundo plano. Quando estiver concluída, a tarefa é inserida na fila de execução [Tilkov e Vinoski 2010]. Tal funcionamento permite a criação de aplicações dinâmicas e facilita a utilização de operações assíncronas.

2.3.2 Redis

Redis, de acordo com Newman (2021), é um banco de dados em memória que é conhecido por sua velocidade e desempenho excepcionais. Ele é amplamente utilizado em aplicações que exigem alta velocidade de acesso a dados, como sistemas de *cache*, análise de dados em tempo real e gerenciamento de sessões. O autor afirma que, no contexto de arquiteturas de microsserviços, o Redis também pode ser utilizado como um *broker* de eventos para gerenciar a comunicação assíncrona entre os microsserviços. Nesse tipo de ambiente, é comum que cada microsserviço execute de forma independente, sem a necessidade de sincronização rígida com outros microsserviços. No entanto, há momentos em que esses microsserviços precisam se comunicar para trocar informações ou

coordenar ações. Nesses casos, conforme estabelece Newman (2021), o Redis pode ser utilizado para gerenciar eventos assíncronos entre os microsserviços. A ferramenta atua como um intermediário para a comunicação entre os microsserviços, permitindo que um microsserviço publique um evento e que outros microsserviços se inscrevam nesse evento para serem notificados quando ele ocorrer.

2.3.3 Docker

De acordo com Nickoloff, Kuenzli e Fisher (2019), o Docker é uma ferramenta para gerenciar *containers*, unidades que fazem o papel de criar uma camada de virtualização com o computador hospedeiro. Cada aplicação pode conter seu próprio ambiente, com rede, arquivos e demais configurações próprias. Dessa forma, cada aplicativo pode carregar consigo um ambiente já pré-configurado com suas próprias dependências. Isso, além de outras coisas, facilita o processo de *deploy* e também o trabalho em equipe, afinal não é necessário que cada desenvolvedor configure sua máquina hospedeira de acordo com as necessidades da aplicação.

2.3.4 MongoDB

Segundo Chauhan (2019), o MongoDB é um sistema de gerenciamento de banco de dados NoSQL, baseado em documentos em formato JSON. É conhecido por sua escalabilidade, flexibilidade e ampla compatibilidade. Ele é frequentemente implementado em aplicações que precisam lidar com grandes volumes de dados, incluindo armazenamento de *logs*, armazenamento de dados em tempo real e análise de dados. Dessa forma o MongoDB oferece uma abordagem mais flexível e dinâmica para o gerenciamento de dados. Ele permite que as aplicações moldem o banco de dados para se adequar às suas necessidades específicas em vez de forçá-las a se adaptarem a um esquema de banco de dados rígido. Além disso, a natureza distribuída do MongoDB oferece alta disponibilidade, escalabilidade horizontal e distribuição geográfica, tornando-se uma opção atrativa para aplicações em grande escala.

3. Trabalhos Correlatos

Os trabalhos correlatos desenvolveram estudos de caso envolvendo microsserviços. Em cada um, foi desenhada, desenvolvida e avaliada uma solução envolvendo essa arquitetura.

O estudo realizado por Alikhujaev (2022) investigou a aplicação de microsserviços em Sistemas de Monitoramento de Pacientes Remotos devido à falta de acesso a profissionais médicos em áreas remotas. O sistema existente foi reconstruído usando princípios de microsserviços e padrões de decomposição. Os resultados sugerem que a implementação baseada em microsserviços aumentou a escalabilidade do projeto, elevando a capacidade de inserção de dados em relação ao sistema monolítico. Além disso, se notou uma melhoria na capacidade de manutenção do programa.

Assunção *et al.* (2021) propuseram uma estratégia para redesenhar recursos legados como microsserviços. Para definir quais recursos deveriam ser redesenhados, foram utilizados quatro critérios: acoplamento, coesão, sobrecarga de rede e modularização de recursos. O estudo mostrou que a estratégia proposta alcançou resultados promissores e sugere uma investigação adicional de seu uso em outros sistemas legados. O principal achado foram as oportunidades de reutilização e personalização que surgiram com a criação de microsserviços.

Bjørndal *et al.* (2021) avaliaram a migração de uma arquitetura monolítica para

uma arquitetura baseada em microsserviços, através de uma metodologia construída a partir de uma revisão da literatura e de uma pesquisa em pequena escala com profissionais de tecnologia da informação. Os dois sistemas foram comparados com base em métricas de latência, como escalabilidade, uso do processador, uso de memória e ocupação da rede. Os resultados mostraram que a arquitetura monolítica oferece melhor desempenho e menor custo para sistemas de pequeno a médio porte, mas a arquitetura de microsserviços apresentou uma relação de escalabilidade significativamente maior, especialmente em ambientes de nuvem.

Bukowiec e Gomulak (2020) desenvolveram uma aplicação baseada no padrão de arquitetura de microsserviços para melhorar a gestão e entender as necessidades dos usuários em relação à grande quantidade de *softwares* licenciados disponíveis. A arquitetura de microsserviços permitiu a separação da aplicação em unidades independentes, facilitando o desenvolvimento, teste e implementação individuais. Entretanto, ela também apresentou desafios, incluindo a necessidade de repensar a organização do código devido à quantidade de serviços criados. Por conta disso, o projeto, que começou com vários repositórios, acabou mudando para um único, por ser mais fácil de manter. A arquitetura também permitiu a liberdade de escolha da tecnologia a depender do tipo de serviço, o que foi essencial no ambiente onde a aplicação estava disposta, com equipes compostas por pessoas com diferentes habilidades e experiências tecnológicas.

Lotz *et al.* (2019) investigaram a viabilidade e os possíveis efeitos da mudança da arquitetura de *software* para uma função complexa de assistência ao motorista ao adotar uma arquitetura baseada em microsserviços. Os resultados mostraram que essa nova abordagem pode reduzir a complexidade e etapas de processos demorados, preparando os sistemas de *software* para futuros desafios, desde que os princípios das arquiteturas de microsserviços sejam cuidadosamente seguidos. Os autores afirmam que a arquitetura de microsserviços pode ser benéfica em aplicações as quais exigem-se mudanças rápidas, escalabilidade e alta modularidade. No entanto, eles alertam sobre a necessidade de avaliar individualmente a arquitetura de *software* para cada projeto em termos de uso, tamanho e escopo, levando em consideração que a arquitetura de microsserviços pode não ser a melhor resposta em todos os casos, principalmente em relação a custos e complexidade.

Lopes (2023) investigou se é possível utilizar uma arquitetura em microsserviços baseada em eventos que automatize programas utilizados por ambientes escolares e AVAs. Essa abordagem demonstra ser promissora, pois a implementação foi bem-sucedida de acordo com as demandas do projeto. O autor afirma que um dos principais desafios foi criar uma arquitetura aberta a novos requisitos, porém ao mesmo tempo rigorosa em relação aos requisitos de negócio. Ele também argumenta que a utilização de *containers* Docker ajudou a facilitar os testes durante o desenvolvimento.

Como demonstrado, os maiores benefícios encontrados pelos estudos foram escalabilidade, publicação independente, modularidade e reusabilidade. Já entre as dificuldades, os principais pontos foram complexidade, alto custo e dificuldade de manutenção.

4. Proposta do Trabalho

As próximas subseções tratarão dos processos de desenvolvimento da aplicação para a escola de dança Aconchego, a qual se localiza na cidade de Vitória da Conquista/BA. Inicialmente, foram realizadas a coleta de requisitos, a elaboração de *mockups* e a

definição da arquitetura. Após isso, se deu início ao processo de codificação da aplicação. Por fim, identificaram-se os principais desafios encontrados e as possibilidades de trabalhos futuros.

4.1 Elicitação dos Requisitos

Inicialmente, realizou-se uma série de entrevistas com o proprietário da escola de dança. Estas entrevistas tiveram como objetivo principal identificar e compreender as principais funcionalidades e características que o *software* deve possuir para atender às necessidades e exigências específicas da escola de dança.

Durante estas entrevistas, uma ampla gama de tópicos foi discutida e explorada, incluindo, entre outros, processos operacionais e fluxos de trabalho atuais, desafios enfrentados pela administração e pelos funcionários, expectativas e preferências dos alunos e metas e aspirações de longo prazo da escola de dança como uma entidade empresarial.

As informações e percepções reunidas por meio destas entrevistas desempenharam um papel crucial na concepção e desenvolvimento do *software*, pois forneceram uma base sólida para o desenvolvimento de uma aplicação que atendesse aos requisitos de negócio da escola de dança.

4.2 Requisitos Funcionais

Requisitos funcionais são as funcionalidades e características que um *software* deve possuir para atender às necessidades e objetivos dos usuários. Esses requisitos descrevem o que o sistema deve ser capaz de fazer e como deve funcionar, de forma clara e objetiva [Valente 2020]. Eles são geralmente especificados em termos de tarefas ou funções que o sistema deve ser capaz de executar. Eles são importantes para garantir que o *software* atenda às expectativas e necessidades dos usuários e cumpra seus objetivos. Os requisitos funcionais RF01 a RF16 podem ser acessados na página da plataforma notion[1].

4.3 Requisitos Não Funcionais

Segundo Valente (2020), requisitos não funcionais são os requisitos que especificam as características e qualidades do sistema que não estão relacionadas com as suas funcionalidades principais. Eles descrevem como o sistema deve ser desenvolvido, mantido, implantado, testado e usado, de forma a garantir a sua qualidade, segurança e desempenho. Esses requisitos incluem aspectos como segurança, arquitetura, usabilidade, compatibilidade, confiabilidade, escalabilidade, entre outros. Os requisitos não funcionais RN01 a RN06 também se encontram na página da plataforma notion [2].

4.4 Mockups

Embora o objetivo deste trabalho não seja o desenvolvimento do *front-end* da aplicação, uma série de *mockups* foram criadas com base nas sugestões recebidas durante as entrevistas realizadas com a escola Aconchego para obter uma compreensão precisa dos requisitos do *software*. Esses *mockups* foram concebidos para fornecer uma representação visual da solução de *software* proposta, com o objetivo de coletar opiniões e sustentar a base para modelagem dos requisitos.

---

[1] Requisitos funcionais:
https://octagonal-fahrenheit-57c.notion.site/Requisitos-funcionais-b6d77ec013e34c4eb7c1f1389c866092
[2] Requisitos não funcionais:
https://octagonal-fahrenheit-57c.notion.site/Requisitos-n-o-funcionais-428c410cce32452bab5d6ad4fb7545c6

Os *mockups* foram criados usando o *software* Whimsical. Eles foram projetados para simular a funcionalidade do *software* proposto e para mostrar as várias características e capacidades que estariam disponíveis para os usuários finais. As telas estão divididas em duas categorias: telas do aluno e telas do administrador. As telas podem ser encontradas no link da plataforma [3].

4.5 Casos de Uso

Os casos de uso especificam os requisitos do sistema. Para cada caso de uso, é descrito o que o sistema deve fazer e como ele deve reagir a entradas específicas do usuário. Eles são usados para capturar os requisitos funcionais da aplicação e para fornecer uma visão geral de alto nível das funcionalidades do sistema. Os casos de uso são importantes para garantir que o sistema atenda às necessidades e expectativas dos usuários e cumpra seus objetivos.

Na Figura 3 é possível ver o diagrama de casos de uso, que mapeia de forma visual as relações entre os diferentes atores do sistema.

Figura 3. Diagrama de casos de uso.

4.6 Arquitetura

Nesta subseção serão abordados os principais módulos do sistema e como eles interagirão entre si a fim de atingir as funcionalidades propostas. É demonstrada na Figura 4 a arquitetura de alto nível da aplicação, a qual é composta por 4 (quatro) microsserviços: pessoas, turmas, notificações e pagamentos. As solicitações chegarão através do API *gateway*, que processará as requisições do *front-end* e encaminhará para o microsserviço designado. O uso do API *gateway* também permitirá a aplicação de políticas de segurança e controle de acesso de forma centralizada. Cada serviço também possuirá um banco de

[3] Telas dos *mockups*: https://whimsical.com/aconchego-5GskMVpKvqfwdzeQSa3GvK

dados para persistir as informações de seus respectivos domínios.

Os serviços publicarão todas as informações que podem ser necessárias no *broker* de eventos. No caso da publicação do evento de pessoa criada, por exemplo, alguns microsserviços podem querer se atualizar ou executar alguma outra rotina assim que uma pesssoa for cadastrada no sistema. Logo, eles poderão ser assinantes do evento de pessoa criada e, assim que o evento for publicado, eles serão notificados e poderão executar a rotina necessária.

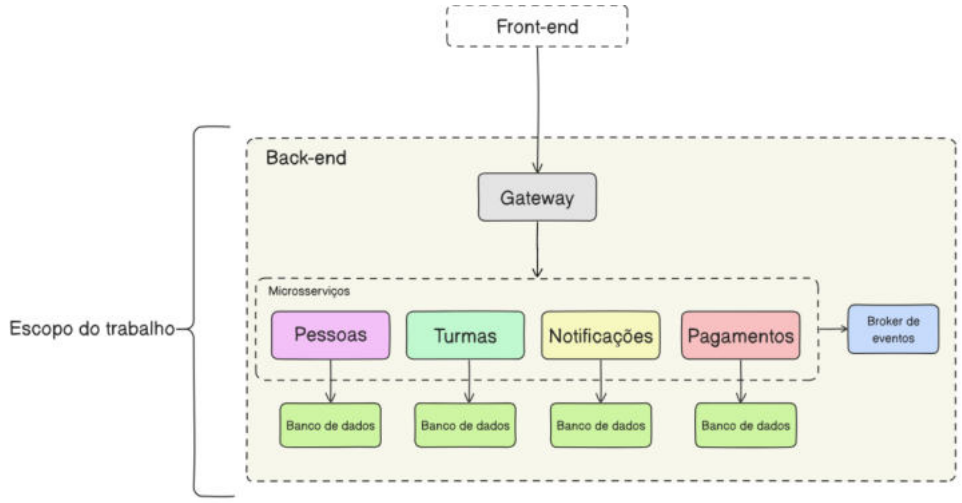

Figura 4. Arquitetura.

O microsserviço "pessoas" desempenha um papel crucial ao lidar com o gerenciamento das informações cadastrais dos usuários e a administração dos diversos papéis existentes no sistema. No que se refere às informações cadastrais dos usuários, ele é encarregado de armazenar e gerir dados relevantes sobre os indivíduos que interagem com o sistema. Esses dados englobam informações básicas, tais como nome, endereço, número de telefone, e-mail e outros detalhes pessoais que possam ser necessários para a identificação e contato dos usuários. Adicionalmente, o serviço "pessoas" também assume a responsabilidade de administrar os papéis desempenhados pelos usuários no sistema, como professor, aluno ou secretária. Cada papel possui diferentes níveis de acesso e permissões no sistema, sendo fundamental que tais informações sejam facilmente acessíveis e possam ser atualizadas conforme necessário.

O microsserviço "turmas" cuida da gestão das informações relacionadas às turmas e aos exames, proporcionando mecanismos para que os administradores monitorem e gerenciem as atividades dos alunos de maneira simplificada. Este serviço é responsável por armazenar e gerenciar informações importantes, como a composição das turmas, o cronograma de aulas, as datas e horários dos exames, os *feedbacks*, bem como outros detalhes relevantes ao funcionamento das turmas. Assim, os administradores poderão acessar facilmente essas informações e tomar decisões informadas sobre o planejamento e organização das atividades. Além disso, o microsserviço "turmas" também terá a função de calcular e registrar a frequência de cada aluno, proporcionando um maior controle sobre a presença nas aulas. O registro de frequência também permitirá a identificação de possíveis problemas, como faltas excessivas e a adoção de medidas para melhorar o engajamento e a participação dos alunos nas atividades de dança.

O microsserviço "notificações" possibilita a distribuição de mensagens da escola de dança para diversos usuários, incluindo alunos, professores e outros membros da

comunidade. Sua principal função é facilitar a comunicação de alertas e notícias relevantes, mantendo os alunos informados sobre o que está acontecendo na escola de dança. Através desse serviço, os alunos poderão receber informações úteis diretamente em seus celulares, como eventos, apresentações, mudanças de horários, dentre outros. Essa comunicação direta e personalizada permite que os alunos se mantenham atualizados e engajados nas atividades da escola de dança. O serviço também pode ser configurado para enviar notificações personalizadas com base nas preferências e necessidades de cada aluno. Isso significa que os usuários podem escolher os tipos de mensagens que desejam receber e os canais de comunicação que preferem utilizar. Essa personalização torna a experiência de comunicação mais eficiente e adaptada às necessidades específicas de cada aluno.

Embora esteja incluído na arquitetura proposta, a implementação do microsserviço "pagamentos" será realizada com o propósito exclusivo de possibilitar o desenvolvimento de trabalhos futuros. A razão para essa abordagem é que a inclusão das diversas funcionalidades inerentes a um serviço de pagamento, como processamento de transações e gerenciamento de contas e segurança, resultaria em um aumento significativo no escopo do projeto atual. Dessa forma, para manter o foco no objetivo principal e evitar a complexidade adicional, as chamadas externas relacionadas ao serviço de pagamento serão apenas simuladas, sem envolver qualquer interação real com sistemas de processamento de pagamentos ou instituições financeiras. Isso permitirá se concentrar no desenvolvimento das tarefas essenciais tendo em mente o escopo em uma aplicação de pequena escala, ao mesmo tempo que deixa a porta aberta para a expansão e aprimoramento do serviço de pagamento em etapas futuras do projeto, conforme as necessidades e prioridades evoluam.

4.7 Criação dos Recursos e Codificação da Aplicação

Inicialmente, foi criado o desenho geral das *APIs*. Após a validação desses desenhos juntamente aos casos de uso, deu-se início à etapa de criação dos recursos necessários para o desenvolvimento. Inicialmente, foram criados e configurados os *containers docker* para a inicialização dos serviços. Através do docker-compose, foram declarados os seguintes containers: 1 instância do Redis, que servirá como *broker* de eventos e 4 instâncias do banco de dados MongoDB, uma para cada microsserviço.

Após essa etapa, iniciou-se o processo de codificação da aplicação. Os serviços foram implementados na linguagem Javascript, utilizando Typescript, e executados através do Node.JS. Eles foram estruturados de maneira modular e escalável para facilitar o desenvolvimento e a manutenção do código. A classe "Server" é a principal unidade de código da aplicação e é responsável por configurar e iniciar o servidor (Figura 5). A estrutura da classe "Server" é semelhante entre todos os serviços. Primeiro, a aplicação é configurada durante a instanciação da classe, com a definição da porta e das funções para a configuração de *middlewares*, rotas e assinantes.

Os *middlewares* são utilizados para interceptar e processar as requisições antes de serem encaminhadas para as rotas. Isso permite a realização de ações como a validação de dados, a autenticação e controle de acesso, dentre outras. Neste caso, são definidos um *middleware* para configurar o acesso de outras aplicações e outro para o tratamento dos dados no formato *JSON* incluído no corpo das requisições.

Figura 5. Código da classe principal "Server".

A seção de configuração dos assinantes vincula eventos específicos às suas respectivas funções. Esses eventos podem ser desencadeados em várias partes da aplicação. Por exemplo, quando algum outro microsserviço publicar um evento de "PESSOA CRIADA", será possível registrar uma ação para esse evento.

A chamada de função para a configuração das rotas associa cada rota da aplicação a um determinado "*handler*", que será o responsável por executar a lógica da rota associada. Ao fim da execução, ela retorna uma resposta ao cliente.

Finalmente, a função é chamada para iniciar o servidor. Quando o servidor estiver apto a receber requisições, uma mensagem é logada no console para indicar o endereço e a porta de escuta. Dessa forma, cada microsserviço foi projetado para ser eficiente, organizado e fácil de gerir. O repositório da aplicação com a implementação completa está disponível no GitHub[4].

5. Considerações Finais

Ao analisar qualitativamente a implementação dos microsserviços no contexto da aplicação para a escola Aconchego, é possível levantar algumas considerações positivas e negativas.

Primeiramente, os microsserviços requerem uma significativa capacidade de infraestrutura e uma alta complexidade operacional. Cada serviço teve que ser implantado e monitorado individualmente, tendo requisitos de infraestrutura diferentes. Além disso, a comunicação entre os serviços precisou ser estabelecida e protegida, o que foi tecnicamente desafiador. A complexidade pode ser notada na Figura 6, que demonstra a grande quantidade de serviços necessários para rodar cada microsserviço. Há de se notar, porém, que tal complexidade pode ter sido acentuada neste trabalho por possuir apenas um desenvolvedor. Em ambiente de trabalho com uma quantidade maior de desenvolvedores, a complexidade pode se tornar um empecilho menor.

[4] Código fonte da aplicação: https://github.com/Aconchego012/aconchego-app

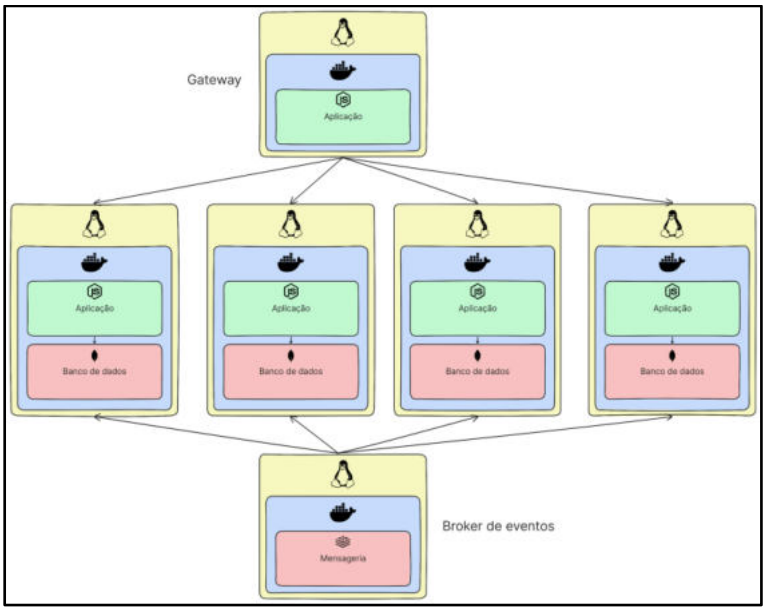

Figura 6. Complexidade operacional.

Em segundo lugar, a aplicação final ficou pouco flexível às mudanças, pois as fronteiras de cada serviço tiveram que ser definidas com base nos levantamentos realizados na documentação inicial do processo. Tais levantamentos estão limitados às regras de negócio e não discutem aspectos relacionados ao uso prático da aplicação. Isso pode se tornar um problema caso a coleta de requisitos não tenha sido adequada ou caso os requisitos de negócio mudem com frequência. Em terceiro lugar, a manutenção dos serviços pode ser desafiadora, pois atualizar ou corrigir um serviço pode ter implicações em outros serviços. Isso pode ser visualizado na Figura 7, onde é demonstrado um dos fluxos da aplicação.

Como ponto positivo, é possível citar a modularidade da aplicação, que facilitou a codificação dos microsserviços, já que cada uma tinha que cuidar apenas de suas próprias regras de negócio. Ao levar em consideração o contexto do trabalho e as informações levantadas, é possível dizer que os benefícios de uma arquitetura em microsserviços não superam os desafios. Portanto, para pequenas aplicações, uma arquitetura monolítica pode ser mais adequada. Ela é menos complexa de implementar e manter, e pode proporcionar um melhor desempenho, já que todas as partes do aplicativo estão juntas em um único pacote.

Como trabalho futuro, sugere-se a implementação da versão monolítica desse domínio para que comparações diretas possam ser analisadas juntamente com o contexto do *front-end* desenvolvido. Além disso, como são conceitos com origens semelhantes, seria oportuno também uma implementação utilizando o conceito de *micro-frontends*, para verificar se os resultados são semelhantes aos deste trabalho. Por fim, sugere-se ainda uma aplicação utilizando somente comunicação síncrona entre os serviços, visto que a comunicação assíncrona adicionou alguns níveis de complexidade, como discutido acima.

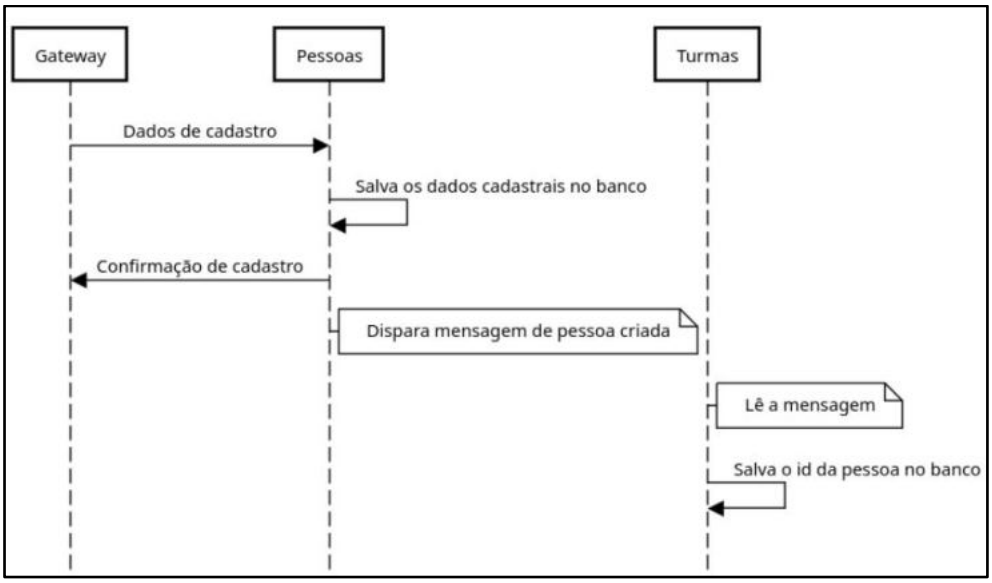

Figura 7. Fluxo para criação de aluno.

Referências

Alikhujaev, A. (2022) *Microservices In IoT-based Remote Patient Monitoring Systems:* Redesign of a Monolith. Trabalho de Conclusão de Curso. University of Twente, Enschede.

Assuncao, W. K. G., Colanzi, T. E., Carvalho, L., Pereira, J., Garcia, A., Lima, M., Lucena, C. (2021) A Multi-Criteria Strategy for Redesigning Legacy Features as Microservices: An Industrial Case Study. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering,* p. 377-387.

Bjørndal, N., Mazzara, M., Bucchiarone, A., Dragoni, N. e Dustdar, S. (2021) *"Migration from Monolith to Microservices: Benchmarking a Case Study"*. The Journal of Object Technology, v. 20, n. 2, p. 3:1.

Bogner, J., Fritzsch, J., Wagner, S. e Zimmermann, A. (2019) Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In: *IEEE International Conference on Software Architecture Companion.*

Bondi, A. B. (2000) Characteristics of scalability and their impact on performance. In: *Proceedings of the 2nd international workshop on Software and performance*.

Bukowiec, S. e Gomulak, P. T. (2020) *"Winventory: microservices architecture case study"*. EPJ Web of Conferences, v. 245.

Fowler, M. e Lewis, J. (2014) "Microservices", https://martinfowler.com/articles/microservices.html, [acessado em 18/10/2023].

Glen, A. (2018) "Microservices Priorities and Trends", https://dzone.com/articles/dzone-research-microservices-priorities-and-trends, [acessado em 02/10/2023].

Gupta, L. (2023) "What is REST", https://restfulapi.net/, [acessado em 10/11/2023].

Iturralde, R. (2020) "Event-Based Processing for Asynchronous Communication", https://aws.amazon.com/pt/blogs/architecture/event-based-processing-for-asynchronous-communication/, [acessado em 11/11/2023].

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J. e Tilkov, S. (2018) Microservices: The Journey So Far and Challenges Ahead. In: *IEEE Software*, v. 35, n. 3, p. 24–35.

Knoche, H. e Hasselbring, W. (2019) *"Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany"*. Enterprise Modelling and Information Systems Architectures. v. 14, p. 1-35.

Lindsay, D., Gill, S. S., Smirnova, D. e Garraghan, P. (2021) *"The evolution of distributed computing systems: from fundamental to new frontiers".* Computing, v. 103, n. 8, p. 1859–1878.

Lopes, E. (2023) *Uma arquitetura de microsserviços baseada em eventos para sincronização entre ambientes escolares e AVAs*. Trabalho de Conclusão de Curso. IFBA, Vitória da Conquista/BA.

Lotz, J., Vogelsang, A., Benderius, O. e Berger, C. (2019) Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study. In: *IEEE International Conference on Software Architecture Companion*.

Loukides, M. e Swoyer, S. (2020) "Microservices Adoption in 2020", https://www.oreilly.com/radar/microservices-adoption-in-2020/, [acessado em 20/06/2023].

Newman, S. (2021) *Building microservices:* designing fine-grained systems. 2ª ed. Beijing Sebastopol, CA: O'Reilly Media.

Nickoloff, J., Kuenzli, S. e Fisher, B. (2019) *Docker in action*. 2ª ed. Shelter Island: Manning.

Nygard, M. T. (2012) *Release it! design and deploy production-ready software*. Dallas, Texas Raleigh, North Carolina: The Pragmatic Bookshelf.

Oregon State University (2014) "Determining the Size of IT Projects", https://is.oregonstate.edu/strategic-plan-projects/project-management/starting-is-projects/determining-size-is-projects, [acessado em 20/11/2023].

Papazoglou, M. P. e Van Den Heuvel, W.-J. (2007) *"Service oriented architectures: approaches, technologies and research issues".* The VLDB Journal, v. 16, n. 3, p. 389–415.

Richardson, C. (2018) *Microservices patterns:* with examples in Java. Shelter Island, New York: Manning Publications.

Söylemez, M., Tekinerdogan, B. e Tarhan, A. (2022) *"Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review".* Applied Sciences, v. 12, n. 11, p. 5507.

Tilkov, S. e Vinoski, S. (2010) *"Node.js: Using JavaScript to Build High-Performance Network Programs"*. IEEE Internet Computing, v. 14, n. 6, p. 80–83.

Valente, M. T. de O. (2020) *Engenharia de Software Moderna*. Independente.

Wazlawick, R. (2011) *Metodologia de pesquisa para ciência da computação*. Elsevier.

O48a Oliveira, Luan Rodrigues de

Arquitetura de microsserviços back-end em uma aplicação de pequena escala: estudo de caso escola aconchego. / Luan Rodrigues de Oliveira. – Vitória da Conquista-BA : IFBA, 2023.

15 f.il.: color.

Orientador: Prof. Pablo Freire Matos

Trabalho de Conclusão de Curso (Graduação) – Bacharelado em Sistemas de Informação. Instituto Federal de Educação, Ciência e Tecnologia da Bahia - *Campus* Vitória da Conquista - BA, 2023.

1. Estudo de caso. 2. Aplicação back-end. 3. Microsserviços.
I. Matos, Pablo Freire. II. Título.

CDD: 621.39

Ficha catalográfica elaborada pela Biblioteca do IFBA *Campus* Vitória da Conquista-BA.