MATHEUS AMÂNCIO FERREIRA

COFRINHO: ABORDAGEM PRÁTICA NA CRIAÇÃO DE UM SISTEMA

LAVRAS - MG
2023

MATHEUS AMÂNCIO FERREIRA

COFRINHO: ABORDAGEM PRÁTICA NA CRIAÇÃO DE UM SISTEMA

Relatório técnico apresentado à Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel.

Prof. André Grützmann
Orientador

LAVRAS - MG
2023

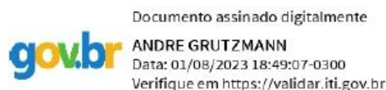
MATHEUS AMÂNCIO FERREIRA

COFRINHO: ABORDAGEM PRÁTICA NA CRIAÇÃO DE UM SISTEMA
COFRINHO: PRATICAL APPROACH FOR A SYSTEM CREATION

Relatório técnico apresentado à Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel.

APROVADO em 19 de Julho de 2023.

Prof Neumar Costa Malheiros
Prof Paulo Afonso Parreira Júnior



Prof. André Grützmann
Orientador

LAVRAS - MG
2023

AGRADECIMENTOS

Agradeço primeiramente a Deus, por me dar forças e suporte todos os dias.
Gratidão a todos os meus familiares que sempre foram um refúgio de proteção, em especial meus pais, Amaury e Aurea e meu irmão Rafael.
Aos amigos que estiveram presentes em toda a minha caminhada, em especial: Augusto Domingues, Daniel Macedo, Fabrizzio Castro, Jean Lopes, João Almeida, Rodrigo Faria e Victor Cabral.
À minha companheira e melhor amiga, Laura Diniz, por todas as risadas, conselhos, ajuda, amor e dedicação.
Agradeço aos docentes e técnicos do Departamento de Ciência da Computação, em especial aos professores Grützmann, pela oportunidade de orientação e conselhos quanto ao projeto, e Neumar, pela amizade e todas as oportunidades.
A todos relacionados com a Universidade Federal de Lavras, por todo o serviço prestado que levaram à este momento.
A todos os meus professores, em especial Hércules Alves, que possibilitaram esta jornada.

RESUMO

A educação financeira é um dos grandes pilares quando se trata da saúde das finanças de qualquer pessoa. Não só saber economizar e gastar, mas também entender como todo o sistema funciona. Ainda que o assunto esteja presente na base curricular do ensino fundamental desde 2017, a minoria das escolas conseguiram adotar o modelo até o momento. Segundo o PISA 2018, cerca de 44% dos estudantes brasileiros estavam abaixo dos conhecimentos mínimos na área, colocando o Brasil em 17º lugar, dentre os 20 países pesquisados. Surgiu então a proposta do Cofrinho, um aplicativo de educação financeira voltado para crianças e jovens. Este trabalho focou-se nos aspectos técnicos do desenvolvimento de uma API HTTP em Node.js e no planejamento de um ambiente em nuvem capaz de suportá-la. Foram empregados conceitos das metodologias ágeis Scrum e Kanban, como forma de seguir padrões de organização bem consolidados no mercado, adaptados a uma equipe pequena. No desenvolvimento, foram aplicados alguns dos padrões e metodologias de software definidos pela metodologia CLEAN e padrão SOLID, como forma de gerar uma base de código bem estruturada e de fácil manutenção. A implementação foi baseada no conceito de MVP, incluindo as funcionalidades mínimas para o produto, envolvendo um módulo de gestão financeira e educacional, permitindo acesso por meio de escolas. A aplicação foi desenvolvida em Typescript e recebeu uma abordagem com múltiplos bancos de dados, sendo o principal o PostgreSQL e o secundário o MongoDB. Ao seguir as técnicas e metodologias propostas, houve o desenvolvimento de um protótipo avançado para o Cofrinho, sendo capaz de cumprir com seus objetivos de MVP.

Palavras-chave: Educação Financeira, API REST, Clean, Solid, Metodologias Ágeis.

ABSTRACT

Financial education is one of the great pillars when it comes to the financial health of any person. Not only knowing how to save and spend, but also understanding how the whole system works. Although the subject has been present in the elementary school curriculum since 2017, few schools have managed to adopt the model so far. According to PISA 2018, about 44% of Brazilian students were below the minimum knowledge in the area, placing Brazil in 17th place among the 20 countries surveyed. Thus arose the proposal of Cofrinho, a financial education application aimed at children and young people. This work focused on the technical aspects of developing an HTTP API in Node.js and planning a cloud environment capable of supporting it. Concepts from agile methodologies Scrum and Kanban were employed as a way to follow well-established organizational standards in the market, adapted to a small team. In development, some of the software patterns and methodologies defined by the CLEAN methodology and SOLID standard were applied, as a way to generate a well-structured and easily maintainable code base. The implementation was based on the MVP concept, including the minimum functionalities for the product, involving a financial and educational management module, allowing access through schools. The application was developed in Typescript and received an approach with multiple databases, with PostgreSQL being the main one and MongoDB being secondary. By following the proposed techniques and methodologies, there was the development of an advanced prototype for Cofrinho, being able to fulfill its MVP objectives.

Keywords: Financial Education, API REST, Clean, Solid, Agile Methodologies.

LISTA DE SIGLAS

API	*Application Programming Interface* (Interface de Programação de Aplicação)
AWS	*Amazon Web Services* (Serviços Web Amazon)
CNDL	Confederação Nacional de Dirigentes Lojistas
DTO	*Data Transfer Object* (Objeto de Transferência de Dados)
ENEF	Estratégia Nacional de Educação Financeira
GB	Gigabyte
HTTP	*Hypertext Transfer Protocol* (Protocolo de transferência de hipertexto)
ID	Identificador
LGPD	Lei Geral de Proteção dos Dados Pessoais
MB	Megabyte
MVP	*Minimum Viable Product* (Mínimo produto viável)
NPM	*Node Package Manager* (Gerenciador de pacotes node)
OCDE	Organização para a Cooperação e Desenvolvimento Econômico
ORM	*Object Relational Mapper* (Mapeador objeto relacional)
PEIC	Pesquisa Nacional de Endividamento e Inadimplência do Consumidor
PISA	Programa Internacional de Avaliação de Estudantes
REST	*Representational State Transfer* (Transferência de Estado Representacional)
SPC	Serviço de Proteção ao Crédito
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
vCPU	*Virtual Central Processing Unit* (Unidade de Central de Processamento Virtual)

LISTA DE ILUSTRAÇÕES

Figura 1 - Estrutura de pastas..32
Figura 2 - Fluxograma de processo de desenvolvimento..33
Figura 3 - Modelo da aplicação em arquitetura em 3 camadas...38
Figura 4 - Modelo do banco de dados completo...47
Figura 5 - Modelo do módulo de usuários..48
Figura 6 - Modelo do módulo de escolas..49
Figura 7 - Modelo do módulo de cofrinhos...50
Figura 8 - Modelo do módulo educacional...51
Figura 9 - Esquema de validação da collection relativa às atividades *TrueOrFalse*................52
Figura 10 - Modelo de criação de collections no banco MongoDB....................................52
Figura 11 - Cadastro de estudante...61
Figura 12 - Registro de Transações...62

LISTA DE TABELAS

Tabela 1 - Requisitos funcionais.. 35
Tabela 2 - Requisitos não-funcionais... 36
Tabela 3 - Relação das máquinas potenciais para o sistema.. 40
Tabela 4 - Relação das máquinas para maior demanda..41
Tabela 5 - Relação das máquinas para maior demanda..42
Tabela 6 - Backlog das sprints...43
Tabela 7 - Tabela de entidades...54
Tabela 8 - Rotas HTTP...59
Tabela 9 - Padrão de respostas.. 62

SUMÁRIO

1 INTRODUÇÃO.......... 12
2 REFERENCIAL TEÓRICO.......... 15
2.1 Abordagens ágeis.......... 15
2.2 MVP.......... 16
2.3 Práticas de programação.......... 17
2.4 NODEJS + TYPESCRIPT.......... 18
2.5 ORM.......... 19
2.6 Banco de dados.......... 19
2.7 API HTTP RESTful.......... 21
2.8 Arquitetura do sistema.......... 21
2.9 Uso de contêineres.......... 22
2.10 Kubernetes.......... 23
3 METODOLOGIA.......... 24
3.1 Gerenciamento de projeto.......... 24
3.2 Princípios de desenvolvimento.......... 25
3.3 Processo de levantamento de requisitos.......... 27
3.3.1 Workshops.......... 27
3.3.2 Cenários.......... 28
3.3.3 Validação.......... 28
3.3.4 Documento de requisitos.......... 28
3.4 Processo de modelagem de banco de dados.......... 29
3.5 Processo de design de software.......... 30
3.6 Definição da metodologia de desenvolvimento.......... 33
4 DESENVOLVIMENTO DA APLICAÇÃO.......... 34
4.1 Detalhamento da aplicação.......... 34
4.2 Análise de requisitos.......... 34
4.3 Arquitetura do sistema.......... 37
4.3.1 Detalhamento da arquitetura.......... 38
4.4 Definição da infraestrutura.......... 39
4.4.1 Servidor da aplicação.......... 39
4.4.2 Servidor de banco de dados.......... 41
4.5 Backlog das sprints.......... 43
4.6 Ferramentas e bibliotecas.......... 44
4.7 LGPD.......... 45
4.8 Banco de dados.......... 46
4.8.1 Módulo de usuários.......... 47
4.8.2 Módulo de escolas.......... 48
4.8.3 Módulo de cofrinhos.......... 49
4.8.4 Módulo educacional.......... 50

4.9 Cofrinho API **52**
5 RESULTADOS **54**
5.1 Entidades **54**
5.2 Rotas da API **59**
6 CONCLUSÃO **63**
REFERÊNCIAS **65**

1 INTRODUÇÃO

O projeto Cofrinho é uma iniciativa para desenvolver uma plataforma completa de educação financeira para crianças e adolescentes, com foco na implementação em escolas públicas do Brasil. Segundo a OCDE (2019), com os resultados do PISA 2018, há um déficit enorme no país com relação ao letramento financeiro básico. Isso pode ser percebido facilmente ao observar o número de endividamentos no Brasil. Segundo a Peic (Pesquisa de Endividamento e Inadimplência do Consumidor), em 2022 quase 78% das famílias brasileiras estavam endividadas, a maioria famílias com baixa renda (CNC, 2022).

Uma abordagem de educação financeira, em especial para os jovens, é uma boa alternativa para mudar essa estatística. Ainda segundo a Peic 2022, o perfil de endividamento no Brasil "é de uma mulher, com menos de 35 anos e ensino médio incompleto", evidenciando que é uma prática que deve ser iniciada ainda no ensino fundamental. Dados divulgados pelo CNDL e SPC Brasil indicam que, em março de 2023, quase 22% dos endividados no Brasil eram jovens entre 18 e 29 anos, representando mais de 15 milhões de brasileiros (CNDL Brasil, 2023).

Ao contrário do que pode ser pensado, educação financeira vai além do que apenas saber poupar dinheiro. A definição citada pela ENEF (Estratégia Nacional de Educação Financeira), segundo a OCDE (2005), diz que a educação financeira é:

> o processo mediante o qual os indivíduos e as sociedades melhoram a sua compreensão em relação aos conceitos e produtos financeiros, de maneira que, com informação, formação e orientação, possam desenvolver os valores e as competências necessários para se tornarem mais conscientes das oportunidades e riscos neles envolvidos e, então, poderem fazer escolhas bem informadas, saber onde procurar ajuda e adotar outras ações que melhorem o seu bem-estar. Assim, podem contribuir de modo mais consistente para a formação de indivíduos e sociedades responsáveis, comprometidos com o futuro.

O Cofrinho é uma plataforma móvel de educação que une escolas, alunos e qualquer jovem que deseje receber um letramento financeiro baseado numa abordagem gamificada. Ele é composto por um módulo educacional, onde os jovens recebem materiais e atividades para aprenderem mais sobre educação financeira, um módulo de gestão, onde eles podem criar e gerenciar seus cofrinhos virtuais e um módulo escolar, onde as escolas podem gerir seus alunos e os conteúdos acessados por eles. O sistema se baseia em uma API REST HTTP, objetivo deste trabalho, que se mostra uma forma de centralizar o tratamento dos dados, e padronizar a comunicação entre cliente e servidor. API, sigla para Application Programming Interface (Interface de Programação de Aplicação, em tradução livre), se refere ao contrato de

comunicação entre diferentes serviços e aplicações, explicitando suas regras e modelos, com interfaces bem definidas. As caracterizadas como RESTful foram propostas por Roy Fielding em 2000 e referem-se a um conjunto de princípios que garantem o desacoplamento e padronização de requisições e respostas disponibilizadas por uma aplicação na web (IBM, s.d.). HTTP, diz respeito ao protocolo de comunicação que se baseia.

O desenvolvimento de software com qualidade e previsibilidade depende de um bom planejamento, dando condições de executar um projeto eficientemente e aumentar suas chances de sucesso (PROJECT MANAGEMENT INSTITUTE, 2017). Assim, o projeto foi orientado com uma adaptação das metodologias ágeis Scrum e Kanban, levando em conta a cultura proposta pelas técnicas, e não suas ferramentas específicas (MARQUES, 2021). Nos termos de arquitetura de software, não é diferente. Diversos modelos e técnicas existem com intuito de desenvolver códigos limpos e reutilizáveis. Martin (2009, p. 13) diz que “Se quiser ser rápido, se quiser acabar logo, se quiser que seu código seja de fácil escrita, torne-o de fácil leitura”.

Três populares metodologias nesse sentido foram propostas por Robert “Uncle Bob” Martin: arquitetura limpa (2017), código limpo (2008) e princípios SOLID (2000). Cada uma em seu escopo, definem boas práticas a serem implementadas em qualquer aplicação, que garantem que o código seja bem planejado, organizado, limpo, reutilizável e sólido (no sentido de bem estruturado e imutável). Estes modelos foram adaptados para encarar a realidade do projeto e aplicados conforme os capítulos seguintes descrevem.

O sistema foi desenvolvido em Node.js, ambiente Javascript fora dos navegadores, e Typescript, formando uma combinação popular e crescente no mercado de tecnologia da informação (GITHUB, 2022). A base utilizada no desenvolvimento foi o framework NestJS. Como uma forma de persistência, a solução de banco de dados escolhida para armazenar os dados do Cofrinho foi híbrida entre sistema SQL e NoSQL, utilizando PostgreSQL e MongoDB. Como forma de gerenciar ambos estes sistemas, foi utilizado uma popular biblioteca, chamada TypeORM. O sistema possui um mecanismo de testes unitários, baseado na biblioteca Jest.

O presente trabalho está relacionado ao trabalho de Victor Gustavo Cabral Rodrigues, sobre a matrícula 201820266, nomeado “PLANO DE NEGÓCIOS: CRIAÇÃO DE UMA EDTECH DE EDUCAÇÃO FINANCEIRA PARA CRIANÇAS E ADOLESCENTES”, projeto empreendedor que detalha as especificações do projeto Cofrinho como um todo, além de incluir questões ligadas à inovação e a colocação do produto no mercado. Como um produto, o Cofrinho já conta com uma interface mobile android, a qual será atualizada, após o

encerramento deste trabalho, incluindo a utilização da API. O Cofrinho é propriedade da startup Cofrinho - Educação Financeira, a qual está incubada pelo Tiradentes Innovation Center (Centro de Inovação Tiradentes), que trabalha com startups do ramo da educação.

Sendo assim, para a realização deste relatório, os objetivos foram estabelecidos como forma de definir um guia para alcançar resultados significativos no desenvolvimento do sistema proposto. Como objetivo geral, criar e documentar todo o processo de desenvolvimento de uma API RESTful para o software educacional Cofrinho, como forma de entregar uma plataforma de educação financeira para jovens, provendo ferramentas para uma plataforma gamificada, enquanto simula o ambiente virtual de um sistema financeiro. Dessa maneira, para o presente relatório técnico, foram traçados objetivos específicos como: Selecionar ferramentas capazes de suportar os requisitos da aplicação; Implementar uma API RESTful que siga as metodologias CLEAN e SOLID e que atenda aos requisitos da aplicação; Planejar e aplicar metodologias ágeis no desenvolvimento da aplicação; Implementar e executar testes unitários sobre o produto desenvolvido; Relatar como as diferentes metodologias e tecnologias selecionadas foram adaptadas para a utilização no projeto.

A educação financeira é um grande desafio no Brasil, e o projeto Cofrinho propõe-se a empenhar um papel na mudança desse cenário. Este trabalho foi realizado com a ideia de documentar e relatar o processo de desenvolvimento do MVP da API do Cofrinho, tornando-se uma forma de prototipa-lo e documentar suas características, princípios e funcionalidades.

Portanto, o trabalho está estruturado com uma seção de referencial teórico que explica termos e conceitos abordados durante a execução do trabalho, familiarizando o leitor, em seguida há toda a metodologia de aplicação utilizada, detalhando os passos tomados e explicando decisões estratégicas quanto às normas e regras definidas no escopo do software construído. Na sequência vem um relato do desenvolvimento, explicando a fundo sobre o sistema e suas particularidades, bem como toda a arquitetura que o rodeia. Ao final, há uma conclusão sobre a elaboração do trabalho, bem como os resultados obtidos.

2 REFERENCIAL TEÓRICO

Esse capítulo tem o objetivo de apresentar os conceitos teóricos e ferramentas utilizadas durante o desenvolvimento do trabalho, de forma a introduzir ao leitor os conhecimentos básicos necessários para o entendimento do material como um todo.

2.1 ABORDAGENS ÁGEIS

No contexto deste trabalho, abordagens ágeis se tratam de um conjunto de metodologias aplicadas no gerenciamento de projeto com objetivo de entregar agilidade e eficiência no desenvolvimento do mesmo. Tratam-se de diferentes métodos e técnicas que, segundo Oliveira e Pedron (2021) tem o objetivo de entregar valor ao cliente de forma rápida e contínua. Larieira, Russo, e Silva(2021) dizem que uma das maiores vantagens dessas metodologias está relacionada à agilidade organizacional que é “a capacidade de uma organização de se adaptar rapidamente às mudanças do ambiente, respondendo às oportunidades e ameaças com flexibilidade e eficiência”.

A abordagem surgiu nos anos 1990, com sistemas iterativos e incrementais. Desde então, as metodologias foram mudando e novos conceitos surgiram. Muitas outras áreas vêm adotando esse sistema, Marques (2021) indica que qualquer contexto onde haja incerteza, mudanças e alta complexidade de operações podem receber adaptações das metodologias ágeis de TI. Porém, é necessária cautela e análise antes da aplicação, visto que Oliveira e Pedron (2021) pontuam em seu estudo que empresas públicas e similares, por seu caráter mais rígido, enfrentam dificuldades na aplicação desse tipo de método.

A abordagem proposta neste trabalho foi inspirada no conceito de Scrum e Kanban, e seu modelo será explicado na seção 3.1. Scrum trata-se de uma das técnicas baseada em três pilares fundamentais: transparência, inspeção e adaptação (SCHWABER, SUTHERLAND, 2017). Seu processo se baseia na elaboração e execução de ciclos incrementais, denominados Sprints, onde a equipe trabalha em conjunto para planejar, executar, revisar e aprimorar o produto. A metodologia tem “a pretensão de agregar valor perceptível no produto final” (APARECIDO CARNEIRO et al, 2020, p. 164) a cada ciclo, baseando-se nos feedbacks do cliente. Kanban, por sua vez, originou-se do modelo de gestão Toyota (LAGE JUNIOR; GODINHO FILHO, 2008) e é muito utilizado no gerenciamento de times. Geralmente, “as tarefas do time de software são representadas por cartões [...]. O status de cada tarefa é indicado de acordo com cada coluna do quadro que o cartão está localizado.” (PEREIRA, 2021) e as nomenclaturas das colunas podem variar de modelo para modelo. Em suma, o time

descreve suas tarefas por meio de cartões no kanban e seu status vai sendo alterado conforme ele é trocado de coluna, cumprindo todo o ciclo de vida das tarefas.

Todo projeto, seja de desenvolvimento de software ou de outra área, demanda planejamento e gerência para manter um padrão de qualidade adequado. Stopa e Rachid (2019), afirmam que a falta de métodos de gerenciamento podem acarretar em problemas de qualidade no projeto final. Nesse sentido, optou-se por implementar uma abordagem ágil baseada nos conceitos de Scrum e Kanban, seguindo uma abordagem incremental, com avaliações constantes. Os papéis definidos no trabalho de Oliveira e Junior (2015 apud STOPA e RACHIDO, 2019), foram utilizados como base, e modificados da seguinte forma:

- PO (Product Owner, “dono” do produto): responsável por entender as demandas do produto e transformá-las em tarefas de aspecto técnico para o time;
- Scrum Master (mestre do scrum): responsável por quebrar as tarefas em atividades menores e designá-las ao time de desenvolvimento. Planeja as sprints e define pesos para as atividades;
- Desenvolvedor: responsável por desenvolver o produto, com base nas decisões do Scrum Master;
- Testador: responsável por testar as funcionalidades antes de serem aprovadas. No contexto deste trabalho, o testador também foi utilizado como uma segunda etapa na revisão de código.

2.2 MVP

MVP, ou *Minimum Viable Product* (Produto Mínimo Viável), consiste num modelo de negócios embasado em lançar uma versão simples e funcional de um produto, com características mínimas. Isso é, mantendo apenas o necessário para testar o modelo proposto. Rizardi e Vicente (2020) caracterizam o modelo como uma forma barata de errar e testar.

De acordo com Cooper e Valdeck (2016), a adoção do modelo MVP traz diversos benefícios, como o desenvolvimento ágil e de qualidade, a satisfação do cliente, a coleta de feedbacks, a implementação rápida de melhorias e correções, e até mesmo o estímulo à criatividade. Para implementar o modelo MVP, existem alguns passos que auxiliam no processo (FIESLER et al., 2018):

1. Definir o problema que se quer resolver e a proposta de valor do produto;
2. Identificar o público-alvo e as suas necessidades;
3. Pesquisar o mercado e a concorrência;
4. Escolher as funcionalidades essenciais do produto;

5. Desenvolver e testar o MVP com um grupo de clientes potenciais;
6. Analisar os resultados e as sugestões recebidas;
7. Repetir os passos 5 e 6 até construir um produto satisfatório ou recomeçar do passo inicial.

Como forma de definir os requisitos iniciais do projeto foi necessário realizar uma etapa de levantamento de requisitos. Como evidenciado por Faria (2016), o mercado atual de desenvolvimento de software não se prende a uma única forma de levantar requisitos, mas utiliza uma série de abordagens distintas como forma de aperfeiçoar essa etapa. Segundo ÁVILA e SPÍNOLA (2008 apud, FARIA, 2016), a etapa de levantamento de requisitos é a que menos custa tempo de desenvolvimento, e a que mais introduz erros. É também a mais barata para executar correções.

2.3 PRÁTICAS DE PROGRAMAÇÃO

Boas práticas, metodologias e padrões são necessários para desenvolver softwares com qualidade. É importante basear-se naquilo que já foi utilizado, testado e validado, como forma de não “reinventar a roda”. Mas não somente isso

> aprender a criar códigos limpos é uma tarefa árdua e requer mais do que o simples conhecimento dos princípios e padrões. Você deve suar a camisa; praticar sozinho e ver que cometeu erros; assistir os outros praticarem e errarem; vê-los tropeçar e refazer seus passos; Vê-los agonizar para tomar decisões e o preço que pagarão por as terem tomado da maneira errada. (MARTIN, 2009)

É necessária prática e vivência. Assim, embasar-se na literatura sobre o assunto parece uma boa ideia, ao menos para conseguir uma base onde apoiar-se. E com essa ideia, algumas das mais famosas filosofias para desenvolvimento surgiram, apoiando-se na vivência que grandes nomes da computação, como o próprio Robert Martin, tiveram em suas vidas.

A base para a metodologia *Clean Code*, ou código limpo em tradução livre, como evidencia seu criador, Martin (2009, p.14) “Não basta escrever um código bom. Ele precisa ser mantido sempre limpo”. Ela dita métodos, filosofias e ideias que ao serem aplicados corretamente elevam a qualidade de leitura e escrita de códigos e partem de conceitos simples, como melhorar a nomenclatura de variáveis, até ideias mais complexas, como reduzir a complexidade cognitiva de uma função (em resumo, sua dificuldade de leitura). Antes disso, Robert ainda propôs o que seriam chamados de padrões SOLID, que determinam conceitos para um código sólido, com baixo acoplamento entre recursos, reusabilidade, coesão com a ideia do que um componente deve realizar, entre muitas outras ideias (PAIXÃO, 2019).

A organização do código escrito também é um fator importante para a qualidade do código. Uma filosofia/ metodologia também foi proposta por Martin, em 2017 chamada de *Clean Architecture*, arquitetura limpa em tradução livre. Uma forma de resumi-la seria dizer que "a *Clean Architecture* tenta fornecer uma metodologia a ser usada na codificação, a fim de facilitar o desenvolvimento códigos, permitir uma melhor manutenção, atualização e menos dependências." (ZERELLI, 2020) E pode ser aplicada em qualquer ambiente, independente da linguagem, *framework*, ou qualquer outro fator externo. Ela apenas dita limites entre camadas, e como elas se relacionam. (ZARELLI, 2020)

2.4 NODEJS + TYPESCRIPT

A linguagem de programação escolhida para o desenvolvimento do projeto foi o Node.js, uma tecnologia que permite a execução de código JavaScript fora do navegador, em um ambiente de servidor. Ele é baseado no motor V8 do Chrome, conhecido por sua rápida e eficiente interpretação de JavaScript (NODE, s.d). Ele é acompanhado pelo NPM, gerenciador de pacotes Node que é um sistema de bibliotecas, proporcionando mais facilidade de desenvolvimento, aproveitando-se de publicações disponibilizadas em seus repositórios (NPM, s.d.). Toda biblioteca possui índices de popularidade e muitas vezes relatos de segurança, fornecendo uma certa confiabilidade. Segundo pesquisas de uso realizadas pela W3Techs (2023), o Node.js é o mais utilizado em servidores web de sites com grande tráfego, e é utilizado por gigantes da tecnologia, como Twitter, Netflix e GitHub.

Por sua vez, o TypeScript é uma linguagem que estende o JavaScript, adicionando a capacidade de definir tipos estáticos para as variáveis. Desenvolvido e mantido pela Microsoft, é um superconjunto de JavaScript, o que significa que todo seu código é válido em TypeScript (MICROSOFT, s.d.). Uma das suas principais vantagens é tornar o código mais seguro e robusto, prevenindo muitos erros antes mesmo da execução. De acordo com Zammetti (2020), o TypeScript representa uma forma de aprimorar o JavaScript, tornando-o mais adequado para o desenvolvimento de aplicações web modernas e complexas. A pesquisa Octoverse do GitHub (2022) relata o Javascript e Typescript como as linguagens #1 e #4, respectivamente, no ranking das mais populares, fato comprovado pelo StackOverflow (2022) em sua pesquisa anual de 2022. Além disso, esta última está relacionada como a quarta linguagem mais adorada pelos desenvolvedores, segundo a mesma pesquisa.

Os índices de popularidade e utilização em grandes sistemas, com altíssimo volume de tráfego diário, além da preferência pessoal dos desenvolvedores, foram os fatores decisivos na escolha das tecnologias.

2.5 ORM

Os ORMs (Object Relational Mapping ou Mapeamento Objeto-Relacional) são ferramentas que facilitam a interação entre programas orientados a objetos e bancos de dados relacionais. Felipetto (2020) caracteriza os ORMs como “uma estratégia para conversão de objetos em memória, para banco de dados relacionais”, e Barsotti e Gibertoni (2020) descrevem como “uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que eles representam”. Proporciona aumento de produtividade e, somado ao fato de muitos desenvolvedores não se sentirem à vontade para escrever SQL, garantiu o crescimento do uso da técnica (SATO, 2013 apud BARSOTI; GIBERTONI, 2020).

Um exemplo de uso dos ORMs é no desenvolvimento de APIs REST, onde eles podem ajudar a gerenciar as operações de banco de dados, validações, associações e transações envolvidas (SANTOS; SANTOS; SILVA, 2018). Um princípio fundamental ao usar um ORM é monitorar o desempenho das consultas e otimizar o código quando necessário (AMBLER, 2003), pois o código gerado pelos ORMs nem sempre é o mais eficiente. Em alguns casos, é preciso equilibrar a velocidade de uma operação com a velocidade de entrega e tomar decisões com base nesta relação.

Existem diversas bibliotecas que entregam ORMs, em diversas linguagens de programação, como o Hibernate (Java), Dapper (C#), DjangoORM (Python) e TypeORM (Node.js). O ORM selecionado para a plataforma foi o TypeORM, solução de código aberto lançada em 2016 que foi desenvolvida especialmente para Typescript. Numa comparação feita por um concorrente, Prisma (2022), a biblioteca foi descrita como madura, popular, bem estabelecida e flexível.

2.6 BANCO DE DADOS

SGBDs (Sistemas de Gerenciamento de Banco de Dados) são softwares que permitem criar, manipular e gerenciar bancos de dados de forma eficiente e segura. Conforme Oliveira et al. (2018), "os SGBDs oferecem uma interface para consultar, inserir, atualizar e excluir os dados, além de fornecer recursos como controle de acesso, backup, recuperação, transações, integridade e consistência dos dados".

Bancos de dados na nuvem são bancos de dados criados, implementados e acessados em um ambiente de nuvem, que consiste em recursos computacionais virtualizados e distribuídos que podem ser acessados pela internet (OSTI; PEREIRA, 2021). Uma das

principais vantagens dos bancos de dados na nuvem em relação aos bancos de dados convencionais é a sua disponibilidade, especialmente em modelos como banco de dados como serviço (DBaaS) (ORACLE, 2020), nos quais toda a infraestrutura é mantida e garantida por um provedor terceirizado. No entanto, também existem desafios e limitações, como segurança e privacidade dos dados, dependência do provedor, qualidade do serviço, legislação e regulamentação (SOUZA et al., 2017).

Bancos de dados podem ser caracterizados como relacionais (comumente chamados de SQL) ou não apenas relacionais (NoSQL). Bancos relacionais tem como característica seu modelo bem estruturado baseado em tuplas que descrevem os relacionamentos dos dados, seja numa tabela (onde linhas e colunas se relacionam para explicar uma entidade), seja entre tabelas (onde duas colunas se relacionam para explicar uma ligação entre elas) (KNIJNIK, 2022). Souza e Oliveira (2022) indicam que bancos NoSQL implementam diferentes modelos para armazenagem de dados, se desprendendo das linhas e colunas. Bancos baseados em documentos são um exemplo disso, onde organizam os dados por meio de documentos com identificadores únicos.

Para modelar os dados do sistema, foram selecionadas duas bases de dados, PostgreSQL e MongoDB. Esta escolha foi tomada para a modelagem de uma parte da aplicação, que é muito complexa e prioritária. Uma má implementação em SQL poderia levar a problemas de performance no futuro, assim, optou-se por utilizar uma alternativa NoSQL como forma de implementar com agilidade e, posteriormente, analisar melhor o esquema dos dados, partindo para uma implementação mais sólida.

O PostgreSQL foi selecionado por conta de sua popularidade e robustez. Segundo estudo elaborado por Couto et al. (2022), ele tem a característica de trabalhar muito rapidamente com operações de leitura, e tem menos desempenho na escrita, como esperado de um banco relacional. Knijnik (2022), comparando o PostgreSQL com o MongoDB, pontua que ao guardar informações muito requisitadas ou que se relacionam com outras, o primeiro se mostra mais indicado.

O MongoDB foi selecionado por sua popularidade, robustez e dinamismo. O mongo é um banco não estruturado orientado a documentos, indicado quando a aplicação demanda flexibilidade e facilidade em montar consultas, além de funcionar bem com quantidades massivas de dados (HIGOR, 2014 apud SOUZA; OLIVEIRA, 2019).

2.7 API HTTP RESTful

Uma API, em tradução livre, Interface de Programação de Aplicação, nada mais é do que um conjunto de regras que regem a comunicação entre dois softwares (AWS, c2023). Red Hat (2020), exibe uma vantagem do modelo, já que "não é necessário saber todos os detalhes sobre o armazenamento em cache, como os recursos são recuperados ou qual é a origem deles.".

Por sua vez, o protocolo HTTP dita um modelo de comunicação baseado na transferência de hipertexto, que é a base para a internet atualmente. Ele faz parte da camada de aplicação (CLOUDFLARE, c2023). O protocolo define métodos, ou verbos, que ditam as ações requisitadas por meio dele, por exemplo GET (pegar) ou POST (postar). Ele contém ainda uma série de informações valiosas para as requisições e respostas, indicando cabeçalhos, sessões, status, dados, etc.

Estes dois termos caminham juntos há muito tempo, viabilizando comunicação entre distintos sistemas, integrados pela internet. Porém, Roy Fielding, em 2000, definiu um modelo, chamado por ele de REST (em tradução livre, transferência de estado representacional), que define "um conjunto de restrições de arquitetura" (RED HAT, 2020). As APIs que seguem essas restrições são chamadas RESTful. Os princípios possuídos por elas ditam que devem ser independentes de estado, ou seja, toda requisição deve conter começo, meio e fim, sem depender de dados entregues requisições passadas; com interface uniforme, independente de quem se comunica, com recursos simples, porém completos e únicos, e também devem ser capazes de armazenar dados em cache (AWS, c2023).

2.8 ARQUITETURA DO SISTEMA

A arquitetura distribuída é um modelo de organização de sistemas de software que consiste em dividir o sistema em componentes ou serviços independentes, executados em diferentes computadores ou processos, que se comunicam por meio de uma rede. Conforme Calegari (2021) afirma:

> Sistemas distribuídos são executados em computadores autônomos e através de redes autônomas. Isso significa que diferentes partes do sistema podem estar sendo executadas em redes diferentes, em computadores com hardware completamente diferentes, sistemas operacionais diferentes e até mesmo escritas em linguagens de programação diversas. (CALEGARI, 2021)

Dentre os modelos mais comuns para a definição de arquitetura de sistemas, está o de n-camadas. Uma das primeiras propostas nesse tema foi elaborada por W. Dijkstra, em 1968,

onde destaca a importância da divisão em camadas hierárquicas para sistemas de grande porte (VALENTE, 2022). A arquitetura de 3 camadas deriva desse modelo e surgiu para resolver os problemas dos mainframes, na década de 1980 (VALENTE, 2022). Ela pode ser explicada como um modelo

> que organiza aplicativos em três camadas de computação física e lógica: a camada de apresentação ou a interface com o usuário; a camada do aplicativo, na qual os dados são processados; e a camada de dados, na qual os dados associados ao aplicativo são armazenados e gerenciados. (IBM, s.d).

Valente (2022) ainda pontua que, a camada do aplicativo pode possuir diversos módulos, seja para organização ou para proteção. Seu principal benefício, segundo a IBM (s.d) eh que “cada camada pode ser desenvolvida simultaneamente por uma equipe de desenvolvimento separada e pode ser atualizada ou ajustada conforme necessário sem impactar as outras camadas”.

Esse modelo busca aumentar a escalabilidade, disponibilidade, tolerância a falhas, heterogeneidade e transparência do sistema, facilitando a manutenção e evolução do software. Tanenbaum e Steen (2007), explicam que os sistemas distribuídos têm como objetivo eliminar gargalos ou pontos únicos de falha em um sistema. Cavalcante (2019) evidencia que um bom modelo arquitetural pode mitigar essas situações.

2.9 USO DE CONTÊINERES

Contêineres são pacotes de software que contém tudo o que é necessário para executar uma aplicação em qualquer ambiente. Eles virtualizam o sistema operacional e isolam os recursos de CPU, memória, armazenamento e rede no nível do sistema operacional, proporcionando aos desenvolvedores uma visão lógica do sistema, isolada de outras aplicações (DOCKER, s.d.). Por conta dessa isolação, um contêiner deve conter todos os requisitos da aplicação.

A portabilidade é um dos principais diferenciais dos contêineres, permitindo que sejam executados em qualquer lugar, desde um laptop pessoal até um data center na nuvem. Isso facilita o desenvolvimento e a implantação das aplicações, eliminando a dependência do ambiente onde elas são executadas. Nomelini e Galante (2021) dizem que outros benefícios relacionados aos contêineres são flexibilidade, segurança e escalabilidade.

Uma das ferramentas mais populares para trabalhar com contêineres é o Docker (Santos at al., 2019), uma plataforma aberta para desenvolver, enviar e executar aplicações usando contêineres. O Docker permite que os desenvolvedores criem imagens de seus

contêineres, que podem ser compartilhadas entre os desenvolvedores ou enviadas para um repositório central chamado Docker Hub, onde podem ser acessadas por qualquer pessoa (DOCKER, s.d.).

O Docker também oferece uma ferramenta chamada Docker Compose, que permite definir e executar aplicações com múltiplos contêineres usando um arquivo YAML. O Docker Compose facilita a orquestração dos contêineres, permitindo especificar como eles devem se comunicar entre si, quais portas devem ser expostas, quais volumes devem ser montados e quais variáveis de ambiente devem ser utilizadas. Além disso, o Docker Compose permite escalar os serviços horizontalmente ou verticalmente com um simples comando (DOCKER, s.d.).

2.10 KUBERNETES

O kubernetes é uma plataforma de código aberto projetada para orquestrar contêineres e gerenciar aplicações distribuídas. “Os contêineres podem ser utilizados em um servidor ao número de milhares, então foi necessário a criação de orquestradores, uma vez que se torna uma tarefa complexa e desnecessária para humanos realizarem” (JENSEN; MIERS, 2021). Sua criação pela Google, com base em sua experiência em sistemas de larga escala, e sua manutenção pela Cloud Native Computing Foundation (CNCF) destacam sua credibilidade (SILVA; SANTOS; ALMEIDA, 2020).

O sistema se baseia am arquivos declarativos, em formato YAML, onde é possível definir todas as especificações dos contêineres e a quantidade de réplicas que devem existir, deixando toda a tarefa de construção, distribuição e monitoramento para o próprio Kubernetes (MARTINS et al., 2019). Apesar de parecer simples, Jensen e Miers (2021) relatam a grande complexidade que um cluster kubernetes pode apresentar.

As vantagens de utilizar orquestradores são descritas por Jensen et al. (2021) “escalabilidade horizontal, tolerância a falhas, entre outras, além de permitir escalonamento e automação flexíveis”. Ashley et al. (2022) demonstra diversas formas de implementar escalonamento horizontal e vertical, além de discutir sobre suas vantagens, como por exemplo na redução de custos e qualidade do serviço (Mao et al; Mao; Humphrey, 2013 apud Ashley, 2017).

3 METODOLOGIA

O presente capítulo tem como objetivo descrever os passos e procedimentos adotados para o desenvolvimento da API do Cofrinho, fornecendo uma visão geral do planejamento, design, implementação e teste do sistema.

3.1 GERENCIAMENTO DE PROJETO

Para a gestão do projeto, utilizou-se alguns conceitos da metodologia Scrum. O time de tecnologia foi composto por 4 pessoas, uma quantidade pequena. Dito isso, a equipe não segue um padrão definido na literatura, mas adapta-se com o possível, visto o tamanho e a falta de especialistas. O cliente, apesar de não constar como um pessoal da equipe, foi indicado e, no contexto do trabalho, trata-se de um dos sócios da empresa, que tem domínio sobre educação financeira e entende do mercado onde o produto se encaixa. Cada membro tem funções que variam conforme o projeto específico, no caso do desenvolvimento do escopo deste trabalho, foram alocados da seguinte forma:

- Bruno Luna: Cliente;
- Victor Cabral: PO e Testador;
- Jean Lopes: Testador;
- Matheus Amancio: Scrum Master e Desenvolvedor.

Conforme especificado pela metodologia Scrum, o time de tecnologia teve ciclos de trabalho organizados por meio de Sprints, as quais as características foram definidas em conjunto com toda a equipe. Primeiro, organizou-se que toda tarefa deve conter: identificador (número único), nome, descrição (texto contendo o que é esperado e como deve ser implementado), requisito relacionado e estimativa (tempo em horas para o desenvolvimento). Depois, as informações das sprints foram definidas. O prazo foi quinzenal, e o peso máximo de cada uma foi definido como 32 horas.

Alguns ritos foram seguidos, como forma de manter o ciclo Scrum funcionando. Todo começo de Sprint foi marcado por uma reunião de planejamento, onde o PO alimentava o quadro com as tarefas requisitadas e o Scrum Master definia prazos e pesos para as atividades definidas. Diariamente aconteceu um detalhamento, assíncrono, das tarefas realizadas naquele dia, mantendo toda a equipe atualizada sobre o andamento das tarefas. Por fim, houveram reuniões de fechamento de Sprint, onde as tarefas finalizadas foram apresentadas ao PO, junto com as falhas, dificuldades e necessidades. Cada Sprint foi organizada com um quadro baseado no quadro de Kanban, e controlada por meio dos quadros de tarefas integrados no GitLab. Os estágios do quadro foram os seguintes:

- Open: Tarefas já documentadas, descritas e prontas para serem utilizadas, porém ainda não associadas com a sprint atual. Caso todas as tarefas de uma sprint sejam encerradas e ainda reste prazo hábil, alguma tarefa dessa lista será associada com a sprint atual. Esse estágio não possui limite de peso (horas de desenvolvimento).
- TO-DO: Tarefas já documentadas, descritas e prontas para serem utilizadas, já relacionadas com a sprint atual, que devem ser cumpridas dentro do seu prazo. O peso máximo é relacionado com o tamanho da sprint (32 horas máximas).
- Doing: Tarefas marcadas como “Em desenvolvimento”. Limitado a uma tarefa por desenvolvedor. Limite compartilhado com o estágio de Test.
- Hold: Tarefas que estão impedidas, seja por dependência a outra tarefa, ou por problemas externos (esperando liberação de um recurso de terceiros, por exemplo). Não há limite de peso para esse estágio.
- Test: Tarefas finalizadas, que entraram em fase de testes. Caso a equipe identifique alguma falha, a tarefa irá retornar para a fase anterior. Limitado a uma tarefa por desenvolvedor. Limite compartilhado com o estágio de TO-DO.
- Done: Tarefas finalizadas e homologadas pela própria equipe. Pronta para validação externa, com a equipe de testes. Limitado a duas tarefas por testador. Caso esse limite seja superado, algum desenvolvedor pode ajudar na fase de testes (a prioridade é a qualidade do produto).
- Rejected: Tarefas com status de rejeitadas pela validação externa. Caso a sprint esteja no final, serão movidas para a próxima, mas caso haja tempo hábil de correção, podem ser movidas para TO-DO. Não há limite de peso para esse estágio.
- Accepted: Tarefas com status de aceita. Ao final da sprint, são apresentadas e por fim movidas para a etapa seguinte. Não há limite de peso para esse estágio.
- Closed: Tarefas finalizadas, que não devem receber mais alteração alguma. Caso algum recurso desenvolvido nela seja falho no futuro, uma nova tarefa será criada, com um link para a antiga, mas sem alterá-la. Não há limite de peso para esse estágio.

3.2 PRINCÍPIOS DE DESENVOLVIMENTO

Independentemente da aplicação desenvolvida, todo software deve contar com metodologias e padrões, com objetivo de gerar um código limpo, reutilizável e testável, em suma, um código de qualidade. Dessa forma, houve uma busca na literatura sobre bons princípios que deveriam ser seguidos durante toda a fase de desenvolvimento.

Nesse processo o time de desenvolvimento foi o responsável pelas escolhas a seguir. A princípio, três famosos conjuntos de boas práticas foram selecionados, e adaptados conforme a necessidade da aplicação, sempre levando em conta a ideia do bom senso, ou seja, aquilo que atrapalhar no desenvolvimento, ou não couber no projeto, deve ser removido. Os princípios do Clean Code (MARTIN, 2009), pregam por códigos mais legíveis, uma vez que programadores passam muito mais tempo lendo que escrevendo. Por outro lado, SOLID, nomeação de Michael Feather ao trabalho de MARTIN (2000), indica sobre a manutenção do código e sua vida útil. Por fim, a Clean Architecture (MARTIN, 2017) prega uma arquitetura de código com separação de responsabilidades.

Os princípios de cada uma delas foram estudados e adaptados para a realidade e condição do projeto, chegando nos seguintes:

- P01 - Convenção de nomenclatura: a nomenclatura de variáveis de classes devem seguir um padrão lógico;
- P02 - Nomes em inglês: o padrão de nomenclatura deve ser numa língua forte, sendo escolhido o inglês;
- P03 - Códigos simples: implementações devem sempre buscar simplicidade e legibilidade;
- P04 - Resolva problemas pela raiz: todo problema deve ser tratado até o final, e não apenas remediado;
- P05 - Código reaproveitável: todo código deve ser reaproveitados, cumprindo com suas funções básicas;
- P06 - Responsabilidade única: classes e funções devem ser utilizados apenas para sua função básica, sem aproveitá-lo para outros fins;
- P07 - Sem comentários desnecessários: comentários devem ser evitados, o código deve ser autoexplicativo;
- P08 - Princípio de substituição de Liskov: todas as classes que implementam uma mesma interface devem poder ser substituídas umas pelas outras;
- P09 - Segregação de interfaces: interfaces devem se ater aos requisitos básicos e não forçar implementações desnecessárias;
- P10 - Inversão de dependências: depender sempre de abstrações e não implementações, deixando o código mais testável e menos acoplado;
- P11 - Injeção de dependências: dependências devem ser injetadas, facilitando trocas de bibliotecas e testes;

- P12 - Não depender de forma direta de bibliotecas e frameworks: todo código externo utilizado deve estar encapsulado, de forma a deixar o código escrito independente;
- P13 - Independente da base de dados: onde e como os dados são armazenados não deve alterar as regras de negócio da aplicação, sendo possível migrar sem dificuldade;
- P14 - Testes limpos, simples e rápidos: os testes devem ser pequenos e exatos, testando curtos segmentos de código, com o mínimo de informação possível.

Todas essas definições foram utilizadas durante o desenvolvimento do MVP da aplicação, e seu uso foi validado em todas as entregas das sprints pelos testadores. Em algumas situações, com prazos sobressalentes, casuais problemas foram relatados e corrigidos, em outras, com prazo mais curto, foram deixadas para a entrega seguinte. Apesar de não ser o ideal, com um time pequeno foi a alternativa encontrada.

3.3 PROCESSO DE LEVANTAMENTO DE REQUISITOS

Foram selecionadas 3 técnicas para essa etapa, *workshops*, cenários e validação. Cada uma delas foi adaptada a realidade do projeto e o modelo de sua implantação se deu da seguinte forma: diversas sessões de *brainstorming* ocorreram para definir o escopo geral da aplicação, seguidas de reuniões para estabelecer melhor os limites da aplicação; após isso, uma rodada de mapeamento de cenários ocorreu, de forma generalista, determinando quais ações e atores estão envolvidos no projeto. Posteriormente, diversas etapas de validação foram realizadas, com o objetivo de eliminar casos desnecessários e encontrar e solucionar erros no modelo descrito.

Um detalhe importante sobre essa etapa, é que nela foi observada a necessidade de trabalhar com um modelo de MVP, reduzindo o tempo de desenvolvimento num geral, e entregando valor de forma mais rápida. Também é necessário notar que essa abordagem se identifica bem com as abordagens ágeis adotadas, uma vez que “No design ágil para inovação social, buscamos errar rápido e errar barato, para chegar mais rápido a uma boa solução. Mesmo com todo o planejamento do mundo, não é possível acertar de primeira” (RIZARDI; VICENTE, 2020, p. 20).

3.3.1 WORKSHOPS

Essa etapa, como já mencionado, contou com diversas reuniões com a equipe. Na primeira delas, o software Cofrinho foi definido, por meio de *brainstorming*, limitando de forma geral, como ele se colocará no mercado. As próximas duas reuniões foram focadas em tangenciar cada uma das funcionalidades que o aplicativo deveria conter, ainda de uma forma

mais simples, com *brainstorming*. A próxima, se deu para estabelecer os limites que o MVP iria respeitar, com uma técnica baseada na construção de cenários, onde a equipe construiu casos que descreviam os limites do sistema.

3.3.2 CENÁRIOS

Com os limites do produto estabelecidos, iniciou-se uma etapa baseada na criação de cenários e fluxos de informações. Durante esse período, a equipe reunida analisou o escopo do MVP, e o separou em ações gerais e atores envolvidos, de forma bem simples, mas que potencializam a escrita de um documento de requisitos.

3.3.3 VALIDAÇÃO

Essa foi a etapa final, antes da escrita dos requisitos da aplicação na etapa de MVP. Nela, a equipe se reuniu duas vezes, em reuniões mais longas, onde cada uma das ações descritas no passo anterior foram questionadas e seu fluxo de informações foi validado, tudo isso por meio de descrição em papel e caneta. Nessa etapa, alguns fluxos alternativos foram identificados e adicionados à documentação. Ainda nessa etapa, mais algumas ações foram retiradas do MVP, por aumentarem muito o custo do desenvolvimento, sem adicionar tanto ao valor de entrega.

3.3.4 DOCUMENTO DE REQUISITOS

O documento de requisitos em questão foi a parte final do levantamento de requisitos. Ele diz respeito à especificação dos cenários, em uma forma simplificada da proposta por Reinehr (2020, p.141). Eles foram organizados por forma de duas tabelas que contém os requisitos funcionais, e não-funcionais respectivamente. Eles apresentam os seguintes dados:

- Requisitos Funcionais
    - Código: valor único que identifica o requisito;
    - Identificação: nome do requisito;
    - Classificação: diz respeito a importância do requisito no sistema;
    - Ator: qual tipo de ator está envolvido;
    - Objetivo: breve descrição do requisito;
    - Variáveis: valores que devem ser informados pelo ator no momento da execução da tarefa.
- Requisitos Não-Funcionais
    - Código: valor único que identifica o requisito;

- Identificação: nome do requisito;
- Classificação: diz respeito a importância do requisito no sistema;
- Descrição: Descrição do requisito.

3.4 PROCESSO DE MODELAGEM DE BANCO DE DADOS

Tendo a documentação do MVP em mãos, o próximo passo foi modelar o banco de dados. Este processo foi o passo inicial para a aplicação dos processos de gerenciamento de projeto propostos. Aqui, iniciou-se a primeira sprint. Em um *brainstorming* do time de tecnologia, cada requisito foi discutido e as entidades relacionadas a eles foram anotadas. Em um segundo momento, elas foram esculpidas em modelos brutos, com papel e caneta. Após mais uma rodada de *brainstorming*, as entidades envolvidas no MVP puderam ser definidas.

Durante a modelagem das tabelas, percebeu-se que uma abordagem híbrida, com um banco baseado em documentos, poderia ajudar na implementação das atividades, uma vez que possuem uma estrutura mais flexível do que os bancos SQL, e permitem uma implementação mais rápida de estruturas complexas, desde que não haja necessidade de relacioná-las com frequência, como indicam os resultados de Knijnik (2022). Este uso híbrido será mais bem detalhado na seção 5.6. Neste momento, optou-se por incluir um banco de dados MongoDB para modelar essas atividades. O aumento da complexidade de manutenção provido por ele, não foi um percalço grande o suficiente para impedir essa nova abordagem, visto que há muita facilidade de implementação oferecida por essa estratégia. Problemas com a definição da estrutura foram resolvidos com regras de validação, que impedem que documentos fora do padrão permitido sejam enviados a uma coleção. Por fim, atividades com estruturas diversas foram alocadas em coleções diferentes, permitindo uma validação consistente. As desvantagens não sanadas por esta alternativa foram decorrentes da integridade e consistência, que ficaram a nível de aplicação.

Todas as estruturas do banco de dados foram definidas por meio de migrações, incluídas no código-fonte da aplicação. Elas são a definição, em código, de como o banco deve ser estruturado, exibindo a data de quando uma solução foi proposta e como ela deve ser implementada. Sempre que houver atualizações, basta que as novas migrações sejam executadas. Assim, há garantia de versionamento entre camada de aplicação e camada de dados. Essa estratégia funciona bem até mesmo no MongoDB, visto que é possível implementar migrações no mesmo, garantindo que as validações funcionem para os modelos propostos.

3.5 PROCESSOS DE DESIGN DE SOFTWARE

Toda aplicação deve seguir uma estrutura para arquitetar seus componentes, tornando a interação entre eles mais organizada e de fácil entendimento. Aplicando alguns conceitos do método ArchDev, em especial os conceitos de 01 a 05 (CAVALCANTE, 2019), foi possível modelar o design do software, com resultados promissores. Dentre os diversos benefícios ao estruturar bem um sistema, podemos citar

> (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidades de cada componente projetado. (1996, SHAW; GARLAN apud CAVALCANTE, 2019)

Quanto à modelagem interna da API, em sua estrutura de código, foi necessário pensar em como aplicar os princípios definidos na seção 3.2, em especial os relacionados a arquitetura limpa (MARTIN, 2017), com passos de identificação de componentes e planejamento de interação. O resultado encontrado pelo time de desenvolvimento foi baseado na proposta da arquitetura em n-camadas (VALENTE, 2022) aplicada a software, organizando os componentes e como eles se relacionam. Em resumo, os componentes de camadas diferentes só podem conversar com camadas próximas. O modelo de comunicação entre elas foi definido de forma que todas as camadas possam conversar internamente e acessar o domínio e os módulos compartilhados. De resto, pode ser detalhado da seguinte forma:

- Camada de Aplicação: comunica com o cliente, e com os serviços;
- Camada de Serviço: comunica com a aplicação e com os repositórios;
- Camada de Repositório: comunica com os serviço e com os dados;
- Camada de Dados: comunica com os repositórios e com os bancos de dados;

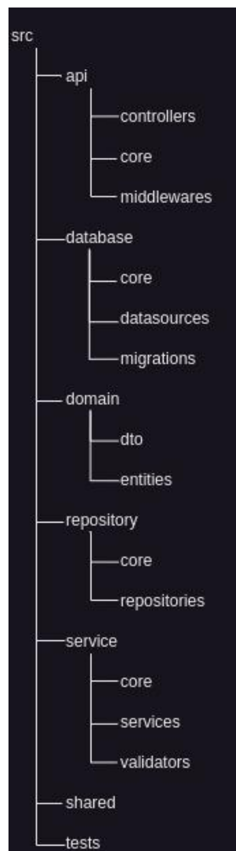
A figura 1 exibe como ficou a organização de pastas, seguindo o design proposto. Segue uma explicação para cada uma delas:

- Source (src): Contém todo o código fonte da aplicação;
  - API: representa a camada de aplicação, a porta de entrada do sistema, que se comunica com os usuários;
    - Controllers: controladores, que indicam as rotas da API, bem como suas regras de acesso. Inclui também as interfaces que os definem. Os arquivos dessa pasta seguem o padrão “nomeDoModulo.controller.ts” e “nomeDoModulo.controller.model.ts”, para arquivos de modelo;

        - Core: arquivos que definem os provedores dessa camada (para injeção de dependências) e a definição dos módulos;
        - Middlewares: funções que são chamadas antes de determinadas rotas, para verificações e definição de valores de sessão. Inclui também seus arquivos de interface. Os arquivos dessa pasta seguem o padrão “nomeDoModulo.middleware.ts” e “nomeDoModulo.middleware.model.ts”, para arquivos de modelo;
    - Database: representa a camada de dados, definição do acesso a ela, bem como as migrações que a alteram;
        - Core: mesmo do tópico anterior;
        - Datasources: arquivos que indicam como conectar-se nas bases de dados. Os arquivos dessa pasta seguem o padrão “nomeDoBanco.datasource.ts”;
        - Migrations: migrações, ou seja, arquivos que contém o passo-a-passo para migrar o banco de dados entre as versões;
    - Domain: camada de domínio, contendo as definições das entidades de DTOs do negócio;
        - Dto: Data Transfer Object, ou Objeto de Transferência de Dados. Utilizados para definir como as camadas se comunicam. Os arquivos dessa pasta seguem o padrão “nomeDoDTO.dto.ts”;
        - Entidades: as entidades, representando as estruturas dos bancos de dados. Os arquivos dessa pasta seguem o padrão “nomeDaEntidade.entity.ts”;
    - Repository: camada de repositório, que contém funções que operam nos bancos de dados;
        - Core: mesmo dos anteriores;
        - Repositories: arquivos que descrevem as classes de repositório, bem como suas interfaces de definição. Os arquivos dessa pasta seguem o padrão “nomeDoModulo.repository.ts” e “nomeDoModulo.repository.model.ts”, para arquivos de modelo;
    - Service: camada de serviço, contendo as regras de negócio da aplicação;
        - Core: mesmo dos anteriores;

    - Services: classes que representam os serviços, bem como seus arquivos de interface. Organizada por pastas internas que representam cada módulo do sistema. Os arquivos dessa pasta seguem o padrão "nomeDoModulo.service.ts" e "nomeDoModulo.service.model.ts", para arquivos de modelo;
    - Validators: classes de validação de domínio para as regras de negócio. Organizada por pastas internas que representam cada módulo do sistema e cada pasta possui um validador de composição único, utilizado por seu serviço correspondente. Os arquivos dessa pasta seguem o padrão "nomeDoModulo.validator.ts" e "nomeDoModulo.validator.model.ts", para arquivos de modelo;
- Shared: módulos compartilhados, bem como encapsulamento de dependências exteriores;
- Testes: módulo de testes. Organizada por pastas internas que representam cada módulo do sistema. Os arquivos dessa pasta seguem o padrão "nomeDoModulo.specs.ts.

Figura 1 - Estrutura de Pastas.



Fonte: Do autor, 2023.

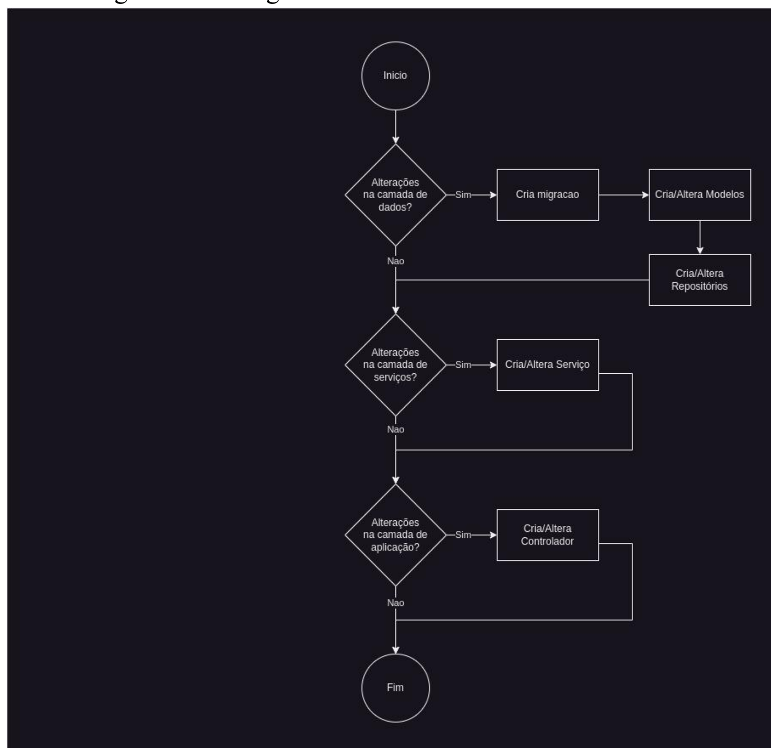
3.6 DEFINIÇÃO DA METODOLOGIA DE DESENVOLVIMENTO

O desenvolvimento da API do Cofrinho seguiu uma metodologia bem definida, respeitando uma estrutura lógica construída para satisfazer todos os requisitos metodológicos definidos. Como forma de satisfazer o princípio P13, optou-se por trabalhar com um ORM, deixando que ele fizesse a camada de adaptação entre diferentes bancos de dados. Aproveitando-se disso, cada alteração no banco de dados foi descrita por meio de migrações. Nesse sentido, criou-se uma camada de interfaces, contendo os modelos descritos no banco. É importante ressaltar que com isso, a aplicação passa a ferir P12, porém os ganhos desta alternativa se destacam bem mais que as perdas.

Como forma de garantir P05 e P06 na camada de dados, foi feito o uso de repositórios, classes que efetivamente manipulam o banco de dados. Cumprindo P10, P11 e P12, eles foram injetados nos locais que dependiam dessa habilidade. A camada de serviços, responsável por aplicar as regras de negócio e processar informações, foi montada no mesmo modelo, com isso, garantindo testes no padrão P14, visto que suas dependências poderiam ser simuladas com objetivo de testar as regras definidas. Toda a definição de rotas de acesso aos usuários foi feita na camada de aplicação por meio de controladores, estruturas que consomem os serviços e garantem a aplicação das permissões.

Todo novo recurso da aplicação foi desenvolvido seguindo o fluxograma da figura 2.

Figura 2 - Fluxograma de Processo de Desenvolvimento.



Fonte: Do autor, 2023.

4 DESENVOLVIMENTO DA APLICAÇÃO

Essa seção relata o conceito geral do projeto e sobre desenvolvimento da aplicação, partindo de decisões arquitetônicas e de modelagem até a implementação da API em ambiente local. Todo o processo envolvido já fora explicado na seção anterior, cabendo então detalhar a aplicação e seus requisitos.

4.1 DETALHAMENTO DA APLICAÇÃO

Como já introduzido anteriormente, o Cofrinho é um aplicativo para controle de finanças e ensino de educação financeira para jovens, em especial as crianças do ensino fundamental. Sua comercialização ocorrerá em etapas, iniciando num projeto de parceria com escolas, e partindo para vendas individuais para pais e responsáveis. Ainda no sentido comercial, apenas o módulo de cofrinhos (gestão financeira) poderá ser acessado gratuitamente.

As parcerias com escolas liberam acesso a seus alunos, por meio de códigos de registro, onde eles poderão acessar conteúdos educacionais fornecidos pela equipe do Cofrinho e pela própria escola. O acesso público, para conteúdo de cofrinhos não demanda nenhum tipo de parceria ou pagamento, dependendo apenas de aceitar os termos de uso da aplicação.

As adições futuras, irrelevantes para esse trabalho, mas que merecem ser citadas, envolvem a adição de um ambiente gamificado, com tarefas, recompensas, placares e uma moeda virtual, que poderá ser trocada em itens de personalização, em suma, cosméticos para seu ambiente virtual.

4.2 ANÁLISE DE REQUISITOS

A primeira fase da análise de requisitos foi definir qual o escopo do MVP. Após todas as reuniões com o time, optou-se por implementar os seguintes módulos:

- Autenticação: dados de registro dos usuários e suas credenciais. Inclui também o gerenciamento de sessão;
- Financeiro: criar, alterar e desabilitar cofrinhos, salvando, inclusive, registros de todas as suas transações;
- Escolar: cadastrar escolas e usuários relativos a ela, sejam representantes ou alunos;
- Atividades: cadastrar atividades a serem realizadas por alunos, e incluir uma forma de premiação opcional para a finalização de uma atividade.

Com esses dados em mãos, um processo de aprofundamento desses módulos foi iniciado, com o objetivo de definir quais ações os usuários do Cofrinho poderiam efetuar durante o uso do sistema. A metodologia aplicada foi descrita na seção anterior, e levou ao mapeamento desses requisitos básicos:

- Um usuário deve poder criar uma conta, alterá-la e deletá-la;
- Um usuário deve poder se autenticar e desautenticar;
- Um usuário deve poder criar cofrinhos, com metas para cada um;
- Um usuário deve poder alterar o valor salvo no cofrinho;
- Um usuário deve poder visualizar todas as operações realizadas num cofrinho ao longo do tempo;
- Devem existir usuários com permissões absolutas (mestre);
- Um usuário mestre deve poder cadastrar escolas;
- Uma escola deve poder ter um ou mais representantes;
- Uma escola deve poder ter zero ou mais alunos;
- Uma escola deve poder cadastrar atividades
- Atividades devem poder ter premiações.

Com tudo isso em mãos, bastou seguir o modelo proposto para elaborar as tabelas com os requisitos funcionais e não-funcionais, respectivamente na tabela 1 e tabela 2.

Tabela 1 - Requisitos funcionais.

Código	Identificação	Classificação	Ator	Objetivo	Variáveis
RF-01	Cadastrar Usuários	Essencial	Usuário	Registrar-se no sistema	Nome, email, data de nascimento, senha
RF-02	Efetuar Autenticação	Essencial	Usuário	Autenticar-se no sistema	Email, senha
RF-03	Acesso de Mestre	Importante	Usuário (Mestre)	Acesso completo aos dados do sistema	N/A
RF-04	Criação de Cofrinho	Essencial	Usuário	Criar um Cofrinho	Nome, meta
RF-05	Depositar no Cofrinho	Essencial	Usuário	Depositar uma nova quantia num Cofrinho	Identificador do Cofrinho, valor
RF-06	Sacar do Cofrinho	Essencial	Usuário	Sacar uma quantia de um Cofrinho	Identificador do Cofrinho, valor
RF-07	Visualizar Cofrinho	Essencial	Usuário	Visualizar um Cofrinho, bem como todas as transações relacionadas à ele	Identificador do Cofrinho
RF-08	Listar Cofrinhos	Essencial	Usuário	Visualizar todos os Cofrinhos ativos do usuário, com um	N/A

Código	Identificação	Classificação	Ator	Objetivo	Variáveis
				resumo do mesmo	
RF-09	Atualizar Cofrinho	Essencial	Usuário	Alterar o nome ou a meta de um Cofrinho	Identificador do Cofrinho, nome, meta
RF-10	Desativar Cofrinho	Essencial	Usuário	Desativar um Cofrinho	Identificador do Cofrinho
RF-11	Elevar um usuário a Mestre	Importante	Usuário (Mestre)	Dar acesso de Mestre à outro usuário	Identificador do Usuário
RF-12	Cadastrar uma escola	Essencial	Usuário (Mestre)	Cadastrar uma nova escola, e um usuário responsável para ela	Nome, CNPJ, grupo, telefone, email, senha
RF-13	Cadastrar um usuário para a escola	Importante	Usuário (Mestre ou Escola)	Cadastrar um novo usuário para uma Escola	Identificador da Escola, nome, email, data de nascimento, senha
RF-14	Cadastrar Atividades	Essencial	Usuário (Mestre ou Escola)	Cadastrar uma nova atividade no sistema	Identificador do usuário, nome, tipo e questão
RF-15	Recomendar Atividade	Desejável	Sistema	Dentre as atividades disponíveis, recomendar uma condizente a um aluno	Identificador do Aluno
RF-16	Cadastrar um aluno para uma escola	Essencial	Usuário (Mestre ou Escola)	Cadastrar um novo aluno para uma escola	Identificador da Escola, nome, email, data de nascimento, senha, turma
RF-17	Efetuar uma atividade	Essencial	Usuário (Aluno)	Exibir e efetuar uma atividade, recebendo uma pontuação por isso	Identificador da Atividade, resposta da atividade
RF-18	Cadastrar Premiação	Desejável	Usuário (Mestre ou Escola)	Cadastrar uma premiação pelo cumprimento de uma atividade ou ação no sistema	Identificador do gatilho, premiação
RF-19	Ativar Premiação	Desejável	Sistema	Ao receber um gatilho de premiação, dar a premiação ao usuário indicado	Identificador do gatilho, Identificador do Usuário

Fonte: Do autor, 2023.

Tabela 2 - Requisitos não-funcionais.

Código	Identificação	Classificação	Descrição
RNF-01	Ambiente Cloud	Essencial	A aplicação deve estar apta a funcionar em ambiente cloud
RNF-02	Performance de Operação	Essencial	O sistema não deve levar mais que 10 segundos para

Código	Identificação	Classificação	Descrição
	Síncronas		executar nenhuma operação síncrona
RNF-03	Performance de Operações Assíncronas	Desejável	O sistema deve notificar o usuário quando uma ação assíncrona estiver encerrada, caso haja alguma
RNF-04	Ambiente de Execução	Essencial	A aplicação deve funcionar em sistemas Linux de 64 bits
RNF-05	Uso de Memória	Essencial	A aplicação deve funcionar com menos de 256 MB de Memória
RNF-06	Uso de Processador	Desejável	A aplicação deve ser capaz* de rodar com 250m vCPU
RNF-07	Réplicas	Importante	A aplicação deve ser capaz de funcionar harmonicamente com qualquer quantidade de réplicas
RNF-08	Uptime	Desejável	A aplicação deve ter um Uptime de no mínimo 90%
RNF-09	Restarts	Importante	A aplicação deve ser automaticamente ligada após falhas
RNF-10	Tempo de Build	Desejável	A aplicação deve ter um tempo de build de no máximo 10s

*Deve ser capaz de funcionar num container limitado a essa quantidade de CPU, visto que o uso do processador é controlado principalmente pelo sistema operacional, e no caso do NodeJS, pode escalar para um núcleo completo, a fim de processar com mais velocidade. Fonte: Do autor, 2023.

4.3 ARQUITETURA DO SISTEMA

O Cofrinho é, a priori, um aplicativo para dispositivos móveis, com características que demandam por um banco de dados centralizado, dentre elas, a principal é a existência de controle de conteúdo por parte das escolas envolvidas no processo. Dessa forma, um modelo de arquitetura distribuída mostrou-se adequado para a implementação.

Abordagens distribuídas proporcionam uma vantagem em escalabilidade e em redução de custos, visto que os recursos podem ser gerenciados individualmente, aumentando a escala em momentos de demanda alta, e diminuindo na baixa. A arquitetura definida se baseia no modelo de 3 camadas que, apesar de segundo a IBM (s.d.) “Atualmente, a maioria dos aplicativos de três camadas são alvos para a modernização” , ainda se mostra como uma base sólida para a estruturação de uma arquitetura, visto suas vantagens de escalabilidade, confiabilidade e segurança (IBM, s.d). Optou-se por não trabalhar com microsserviços, principalmente, por questões de complexidade de desenvolvimento e manutenção por uma equipe pequena.

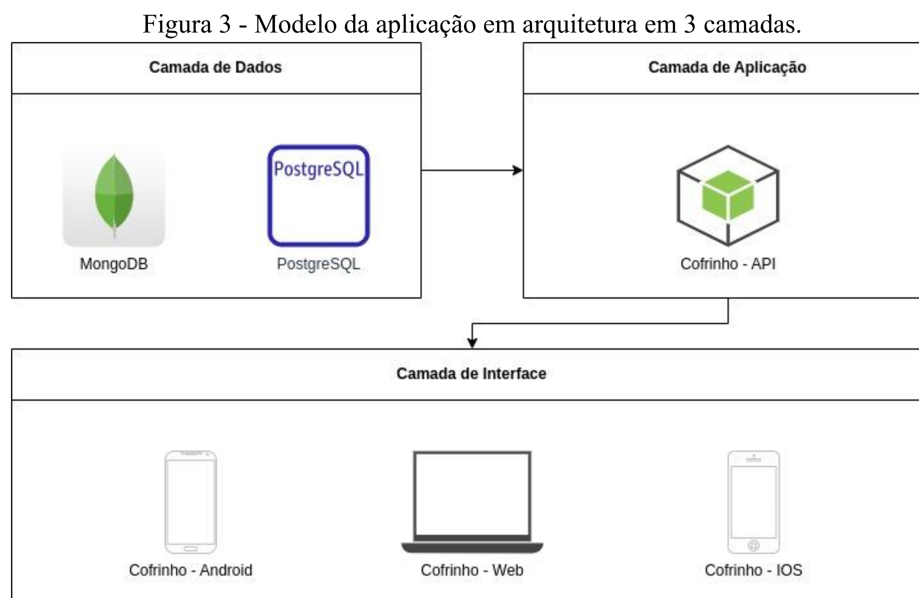
Outra abordagem arquitetural definida foi a de uso de serviços de infraestrutura na nuvem. Boa parte da complexidade envolvida está nas mãos do provedor, como a disponibilidade de máquinas, proteção contra quedas de energia, acesso à internet, etc, como indica a documentação da AWS (s.d.). Outra vantagem está nas soluções oferecidas, como hospedagem de banco de dados personalizada, serviços de fila, mensageria e até mesmo na gerência de clusters. Varella et al (2021) pontua que estes serviços muitas vezes não são

compatíveis entre diferentes servidores, logo é importante uma rápida definição. Com o aumento da necessidade, ainda é possível alugar máquinas mais poderosas e migrar o sistema com passos simples.

Graças a incubação pelo Tiradentes Innovation Center, o Cofrinho recebeu 2000 dólares para serem utilizados com infraestrutura no Google Cloud, potencializando o uso de tecnologias de ponta e vivência do mercado de software.

Para fins de representação, a arquitetura foi dividida em 3 camadas (diferentes das camadas de arquitetura de software), deixando mais clara a participação de cada nó representado. A figura 3 dá uma visualização simplificada da estrutura implementada, ilustrando o fluxo dos dados entre as camadas. As setas indicam como os dados armazenados são direcionados aos usuários, partindo sempre da camada de dados, sendo processados na camada de aplicação e apresentados na camada de interface.

Figura 3 - Modelo da aplicação em arquitetura em 3 camadas.



Fonte: Do autor, 2023.

4.3.1 DETALHAMENTO DA ARQUITETURA

Cada uma das camadas ilustradas na figura 3 representa os pontos chave do sistema, bem como seus participantes. A camada de dados, onde incluem-se os nós PostgreSQL e o MongoDB, desempenha o papel de armazenar as informações pertinentes à plataforma Cofrinho, bem como seus relacionamentos. A camada de aplicação, com o nó "Cofrinho - API", exerce a função de definir as regras de negócio e orquestrar os dados. Qualquer outra solução desenvolvida com objetivo similar será incluída nesta camada. A camada de interface, com os nós "Cofrinho - Android", "Cofrinho - Web" e "Cofrinho - IOS", representa

as interfaces de comunicação com os usuários, seja via navegador web ou acesso por aplicações móveis. Vale ressaltar que essa camada não está no escopo deste trabalho.

4.4 DEFINIÇÃO DA INFRAESTRUTURA

Uma vez definida a arquitetura, iniciou-se uma fase de pesquisa de infraestrutura na nuvem. Como dito anteriormente, por questões de negócio o Google Cloud foi selecionado como provedor de computação em nuvem. A precificação contida nesta seção diz respeito a um levantamento atualizado em junho de 2023.

4.4.1 SERVIDOR DA APLICAÇÃO

Como a aplicação encontra-se em estágio de MVP, a demanda que ela receberá é muito inferior ao esperado em um momento seguinte (fase de comercialização), em especial porque apenas usuários selecionados poderão utilizar a API. Assim, no primeiro momento, o servidor escolhido deve executar uma única instância da aplicação por vez.

Foi feito um levantamento para uma máquina que poderia atender aos requisitos da API, definidos nos requisitos não-funcionais. Assim, faz-se necessário um *hardware* com pelo menos 256MB de memória RAM exclusivo para a API e 250m vCPU. Apesar de não existir um modelo exato para controle de uma margem de erro/segurança, optou-se por trabalhar com 100%, assim, aumentando a necessidade para 512MB. A recomendação de quantidade de RAM para rodar uma instalação de Ubuntu Server, como consta em sua documentação oficial, é de 1GB (CANONICAL, s.d.), somando para 1.5GB. Não há recomendação mínima para quantidade de núcleos. Ainda que seja possível alugar uma máquina com processador compartilhado, a equipe optou por ter mais controle sobre o ambiente, buscando máquinas com no mínimo 1 vCPU.

As máquinas do Compute Engine (sistema de máquinas virtuais na nuvem) são categorizadas por séries, com diferentes recomendações de uso. Dadas as limitações financeiras, optou-se por procurar máquinas baratas e com bom custo-benefício. Segundo a documentação do Google (2023), as instâncias de uso geral com perfis econômico e equilibrado são E2, N2, N2D e N1, as quais foram as únicas consideradas durante a cotação.

Após pesquisa, 2 opções foram consideradas viáveis pela equipe, exibidas na tabela 3. Todos os preços estão em dólar. As configurações base utilizadas no processo de cotação são:

- 1 instância;
- Sistema operacional gratuito;
- Provisionamento regular;

- Máquinas de propósito geral;
- Processadores de 2 threads por núcleo;
- Data Center localizado em “us-east1”;
- 730 horas de uso por mês;
- Sem disco (essa configuração não será utilizada em cenário real, foi proposta para não impactar na pontuação final).

Tabela 3 - Relação das máquinas potenciais para o sistema.

Código	Série	Tipo	Preço	CPU	Fator CPU	Preço/F CPU	RA M	Fator RAM	Preço/F RAM	Pontuação
1	N1	n1-custom-1-2048	$30,71	1	1	30,710	2,00	1,000	30,710	30,710
2	N1	n1-standard-1	$34,67	1	1	34,670	3,75	1,875	18,491	26,580
3	N2	n2-custom-2-4096	$61,42	2	2	30,710	4,00	2,000	30,710	30,710
4	N2D	n2d-custom-2-4096	$53,46	2	2	26,730	4,00	2,000	26,730	26,730

Fonte: Do autor, 2023.

As configurações contam com 1 a 2 vCPUs de processador e 2 a 4GB de memória. O principal fator de decisão foi a pontuação calculada pela média das razões de preço e fator CPU e preço e fator RAM. Esses fatores, por sua vez, são a razão entre a quantidade de recursos ofertada em relação a quantidade buscada. Assim, a máquina escolhida é a #2, que apesar de um pouco mais cara, possui bem mais recursos que a #1. Também foi decidido um plano de emergência, no qual a máquina será alterada para a #4, e mais processos serão iniciados, atendendo maior demanda. Segundo estimativa, essa máquina deve suportar entre 3 e 4 processos com tranquilidade.

Outra pesquisa foi realizada, sob os mesmos parâmetros, para determinar uma máquina que atenda as necessidade de mais usuários, com os resultados na tabela 4. Dessa forma, as necessidades descritas em passo anterior escalam para 6 processos, necessitando de uma configuração com 4GB de memória e pelo menos 2 vCPU. Vale ressaltar que essa pesquisa serve apenas como prevenção, caso haja necessidade de liberar acesso a muitos usuários no sistema. Nessa situação, a máquina escolhida seria a #8 que conta com recursos extras ao requisitado, enquanto apresenta uma pontuação muito satisfatória (segundo na classificação geral).

Tabela 4 - Relação das máquinas para maior demanda.

Código	Série	Tipo	Preço	CPU	Fator CPU	Preço/F CPU	RA M	Fator RAM	Preço/F RAM	Pontuação
1	N1	n1-custom-2-4096	$61,42	2	1	61,420	4,00	1,000	61,420	61,420
2	N1	n1-custom-2-6144	$67,91	2	1	67,910	6,00	1,500	45,273	56,592
3	N1	n1-standard-2	$69,35	2	1	69,350	7,50	1,875	36,987	53,168
4	N2	n2-custom-2-4096	$61,42	2	1	61,420	4,00	1,000	61,420	61,420
5	N2	n2-custom-2-6144	$67,91	2	1	67,910	6,00	1,500	45,273	56,592
6	N2	n2-standard-2	$70,90	2	1	70,900	8,00	2,000	35,450	53,175
7	E2	e2-custom-2-4096	$40,38	2	1	40,380	4,00	1,000	40,380	40,380
8	E2	e2-custom-2-6144	$44,65	2	1	44,650	6,00	1,500	29,767	37,208
9	E2	e2-standard-2	$48,92	2	1	48,920	8,00	2,000	24,460	36,690
10	N2D	n2d-custom-2-4096	$53,46	2	1	53,460	4,00	1,000	53,460	53,460
11	N2D	n2d-custom-2-6144	$59,11	2	1	59,110	6,00	1,500	39,407	49,258
12	N2D	n2d-standard-2	$61,68	2	1	61,680	8,00	2,000	30,840	46,260

Fonte: Do autor, 2023.

Apesar do acesso à máquina ainda não estar disponível no momento, por questões de negócio, ele já foi planejado em ambiente virtual. O servidor contará com uma instalação Ubuntu Server 22.04 LTS limpa, com os seguintes pacotes:

- Node, na versão 18.14.2;
- NVM, na versão 0.39.3;
- PM2, na versão 5.3.0;

A versão do Node foi escolhida com base no desenvolvimento da aplicação. O NPM é um gerenciador de versões do node, seu uso é uma prática utilizada para assegurar possibilidade de fácil atualização no futuro. O PM2 foi escolhido como solução para gerenciamento de múltiplas instâncias da aplicação rodando em paralelo, sendo capaz de criar um balanceador de carga interno. Em um cenário futuro, com acesso a um cluster kubernetes, esse servidor será depreciado e desativado em virtude de sua utilização.

4.4.2 SERVIDOR DE BANCO DE DADOS

A escolha de um banco PostgreSQL foi feita pelo time de desenvolvimento por conta de dois fatores: sua disponibilidade como DBaaS no Google Cloud (como Cloud SQL) e sua excelente performance numa alta demanda de operações, desde que bem configurado, como indica Santos et al (2019). Como consta na documentação, quanto mais memória e poder de processamento disponível, melhor o banco irá responder, porém ele apresenta uma sutil

recomendação a utilizar sistemas com mais 1Gb de RAM (POSTGRESQL, 2023). Sendo assim, foi feita uma pesquisa, como consta na tabela 5, sobre servidores de banco de dados como um serviço (DBaaS). Além disso, alguns outros servidores convencionais foram incluídos, visto que nenhum dos resultados preliminares agradou a equipe.

Tabela 5 - Relação das máquinas para banco de dados.

Código	Série	Tipo	Preço	CPU	Fator CPU	Preço/F CPU	RAM	Fator RAM	Preço/F RAM	Pontuação
1	db	db-lightweight-1	$49,31	1	1	98,620	3,75	0,938	52,597	75,609
2	db	db-standard-1	$49,31	1	1	98,620	3,75	0,938	52,597	75,609
3	db	db-lightweight-2	$79,46	2	1	79,460	3,75	0,938	84,757	82,109
4	db	db-standard-2	$98,62	2	1	98,620	7,50	1,875	52,597	75,609
5	N1	n1-custom-2-4096	$61,42	2	1	61,420	4,00	1,000	61,420	61,420
6	N1	n1-custom-2-6144	$67,91	2	1	67,910	6,00	1,500	45,273	56,592
7	N1	n1-standard-1	$34,67	1	1	69,340	3,75	0,938	36,981	53,161
8	N2	n2-custom-2-4096	$61,42	2	1	61,420	4,00	1,000	61,420	61,420
9	N2	n2-custom-2-6144	$67,91	2	1	67,910	6,00	1,500	45,273	56,592
10	E2	e2-custom-2-4096	$40,38	2	1	40,380	4,00	1,000	40,380	40,380
11	E2	e2-custom-2-6144	$44,65	2	1	44,650	6,00	1,500	29,767	37,208
12	N2D	n2d-custom-2-4096	$53,46	2	1	53,460	4,00	1,000	53,460	53,460
13	N2D	n2d-custom-2-6144	$59,11	2	1	59,110	6,00	1,500	39,407	49,258

Fonte: Do autor, 2023.

A decisão tomada envolve três etapas. No primeiro momento, na etapa de MVP, a equipe optou por gerenciar o banco de dados por si mesma, ou seja, alugando um servidor convencional, configurando o banco e seus recursos. Para isso, será utilizado o servidor #7, que apresenta configuração similar ao DBaaS mais simples (#1 da tabela 5). Quando esse banco começar a falhar por falta de recursos, será feito um upgrade para o #11. Por fim, quando for economicamente possível, o banco será migrado para uma solução mais adequada, como a #3 ou #4. Enquanto o servidor for gerenciado pela própria equipe, ele será um Ubuntu Server 22.04, com o PostgreSQL na versão 15 e suas dependências instaladas. Quando ele for gerenciado automaticamente pelo Google, não há escolha quanto a isto.

A escolha do banco MongoDB se deu pela necessidade de um banco flexível e com rápidas operações de leitura. Infelizmente, ele não está disponível como DBaaS pelo Google Cloud, porém, o MongoDB Atlas, alternativa da própria desenvolvedora do banco, possui um plano gratuito (MONGODB, 2023). Assim, sua aplicação inicial será nesse modelo.

4.5 BACKLOG DA SPRINTS

A tabela 6 apresenta o backlog de todas as sprints, unificadas para melhor visualização. Ela contém dados importantes sobre cada uma e uma definição geral de suas tarefas. Como dito na seção 5.1, cada sprint foi marcado por uma reunião de abertura e fechamento, e apresentações preliminares dos feitos diários.

O peso de cada tarefa foi definido pelo Scrum Master em conjunto dos desenvolvedores e testadores, optando por prazos maiores, para entregas mais completas. Os POs e testadores definiram quais as principais funcionalidades que deveriam ser testáveis, visto que não houve tempo hábil para desenvolver um código com 100% de cobertura.

Tabela 6 - Backlog das sprints

Sprint (#)	Data	Total (horas)	Identificador (#)	Nome	Requisito Relacionado	Estimativa (horas)
1	06/03/2023 a 17/03/2023	32	1	Definição da Arquitetura	RNF-01 RNF-08 RNF-09	12
			2	Planejamento de Infraestrutura	RNF-04 RNF-05 RNF-06	8
			3	Definição do SGBD	-	4
			4	Modelagem do Banco - pt1	-	8
2	20/03/2023 a 31/03/2023	32	5	Modelagem do Banco - pt2	-	4
			6	Construção do Banco	-	10
			7	Modelagem das Entidades	-	4
			8	Construção dos Repositórios	-	12
			9	Autenticação de Usuário - pt1	RF-02	2
3	03/04/2023 a 14/04/2023	32	10	Autenticação de Usuário - pt2	RF-02	6
			11	Validação de Token	RF-02	2
			12	Sessão Única	RF-02	4
			13	Cadastro de Usuários	RF-01	8
			14	Listagem de Usuários	-	2
			15	Edição de Usuário	RF-01	4
			16	Elevação a Usuário Mestre	RF-01	2
			17	Funcionalidades de Usuário Mestre - pt1	RF-11	4
4	17/04/2023 a 28/04/2023	32	18	Funcionalidades de Usuário Mestre - pt2	RF-03	2
			19	Middleware de Timeout	RNF-02	2
			20	Middleware e Validação de	-	6

				Permissionamento		
			21	Criação de Cofrinho	RF-04	4
			22	Listagem de Cofrinhos	RF-06 RF-07	2
			23	Edição de Cofrinho	RF-09	2
			24	Alterar valor no Cofrinho	RF-05 RF-06	2
			25	Desativar Cofrinho	RF-10	2
			26	Cadastrar Escolas	RF-12	8
			27	Criar usuários do tipo Escola	-	2
5	01/05/2023 a 12/05/2023	32	28	Alterar dados da Escola	RF-13	2
			29	Criar usuários do tipo Aluno	-	1
			30	Funcionalidades de Usuário Escola	-	4
			31	Funcionalidades de Usuário Aluno	RF-16	4
			32	Cadastrar Atividades	RF-13	16
			33	Ativação de Premiação ao Final da Atividade	RF-14	1
			34	Recomendação de Atividade	RF-18	2
			35	Cumprir Atividade	RF-19	2
6	15/05/2023 a 26/05/2023	12	37	Migrar para Ambiente Docker	-	8
			38	Middleware de Paginação	RF-15	4

Fonte: Do autor, 2023.

As sprints foram definidas com limite de 32 horas quinzenais, para encaixar na disponibilidade da equipe. O prazo de duas semanas foi escolhido por motivo similar, visto que com menos tempo disponível, as reuniões scrum ocupariam muito tempo, deixando o desenvolvimento significativamente inferior. Em relação às reuniões, cada sprint contou com duas (de abertura e fechamento) com duração média de 30 minutos a uma hora, e comunicação diária assíncrona, para detalhamento do que foi desenvolvido naquele dia. Nos dias onde não houve desenvolvimento, esta comunicação foi mantida, relatando que não houveram tarefas executadas. Em feriados e finais de semana elas foram removidas.

4.6 FERRAMENTAS E BIBLIOTECAS

Como já detalhado em momentos anteriores, a aplicação foi desenvolvida em Node.js, utilizando-se de Typescript. Esta seção detalha as outras ferramentas e bibliotecas relevantes utilizadas no processo de desenvolvimento da API.

A base de todo o projeto se dá no framework NestJS, utilizado para criar e controlar o servidor HTTP, enquanto manuseia suas dependências. Seu uso recomendado baseia-se em muitos dos princípios de desenvolvimento adotados na seção 3.2, em especial os P05, P06, P08, P09, P10, P11 e P12. Apesar de utilizar uma solução desta forma cria uma certa dependência, o que vai contra o P12, este fato pode ser contornado com pouco retrabalho, visto que seu código vai de acordo com muitos outros injetores de dependência, e a definição do servidor HTTP também segue similar a de outras bibliotecas, ou até mesmo de uma solução nativa do Node.js.

Como forma de satisfazer o P13, optou-se por utilizar um ORM, em especial o TypeORM, biblioteca desenvolvida para typescript que possui suporte para ambas as bases de dados utilizadas no sistema. Como forma de evitar o P12, seguiu-se com o padrão de repositórios, isto é, encapsular toda a camada de acesso ao banco de dados em classes especiais. Caso haja necessidade de trocar de biblioteca, sua substituição é simples.

Finalmente, para satisfazer P14, utilizou-se o framework de testes Jest. Infelizmente, não houve uma solução, neste sentido, que satisfaça P12, uma vez que todos os testes escritos dependem explicitamente da biblioteca. Ainda assim, o resultado foi satisfatório, visto que houve grande facilidade no uso.

4.7 LGPD

A LGPD (Lei Geral de Proteção dos Dados Pessoais), dita uma série de regras e hipóteses para coleta e uso de dados, em especial para crianças e adolescentes, público-alvo do Cofrinho. Como uma empresa, é necessário seguir a lei à risca, e todas as informações armazenadas devem estar em conformidade com o requerido. Desta forma, o Cofrinho encontra-se em processo de consultoria com uma empresa externa para adequar-se legalmente, fato este não relevante para o contexto deste trabalho. Todos os itens contidos nesta seção são apenas propostas, e serão avaliadas futuramente.

A primeira responsabilidade da aplicação é a de garantir o consentimento legal dos dados pessoais das crianças, isto é, seu nome, email, data de nascimento e gênero (BRASIL, 2018). Como o principal cliente do Cofrinho são as escolas, planeja-se adicionar um novo campo ao gerar um código de ativação do produto, ditando que quem o gerou é o responsável legal pelo usuário que o ativará, ou possui autorização do mesmo. Isso será um dos tópicos do estatuto do aplicativo, que deverá ser aceito antes de utilizar a aplicação.

Quanto ao uso dos dados, seu estatuto também deve conter um tópico sobre este tema, deixando claro que qualquer dado recolhido poderá ser utilizado como objeto de estudo desde

que devidamente anonimado, isto é, sem conter referência alguma aos dados pessoais do usuário. Um exemplo é sobre o uso do módulo educacional, onde o Cofrinho poderá gerar relatórios sobre o tempo gasto entre atividades, taxa de acerto, entre outros, desde que não segregue os dados por usuário nem divulgue informações sobre eles (BRASIL, 2018).

4.8 BANCO DE DADOS

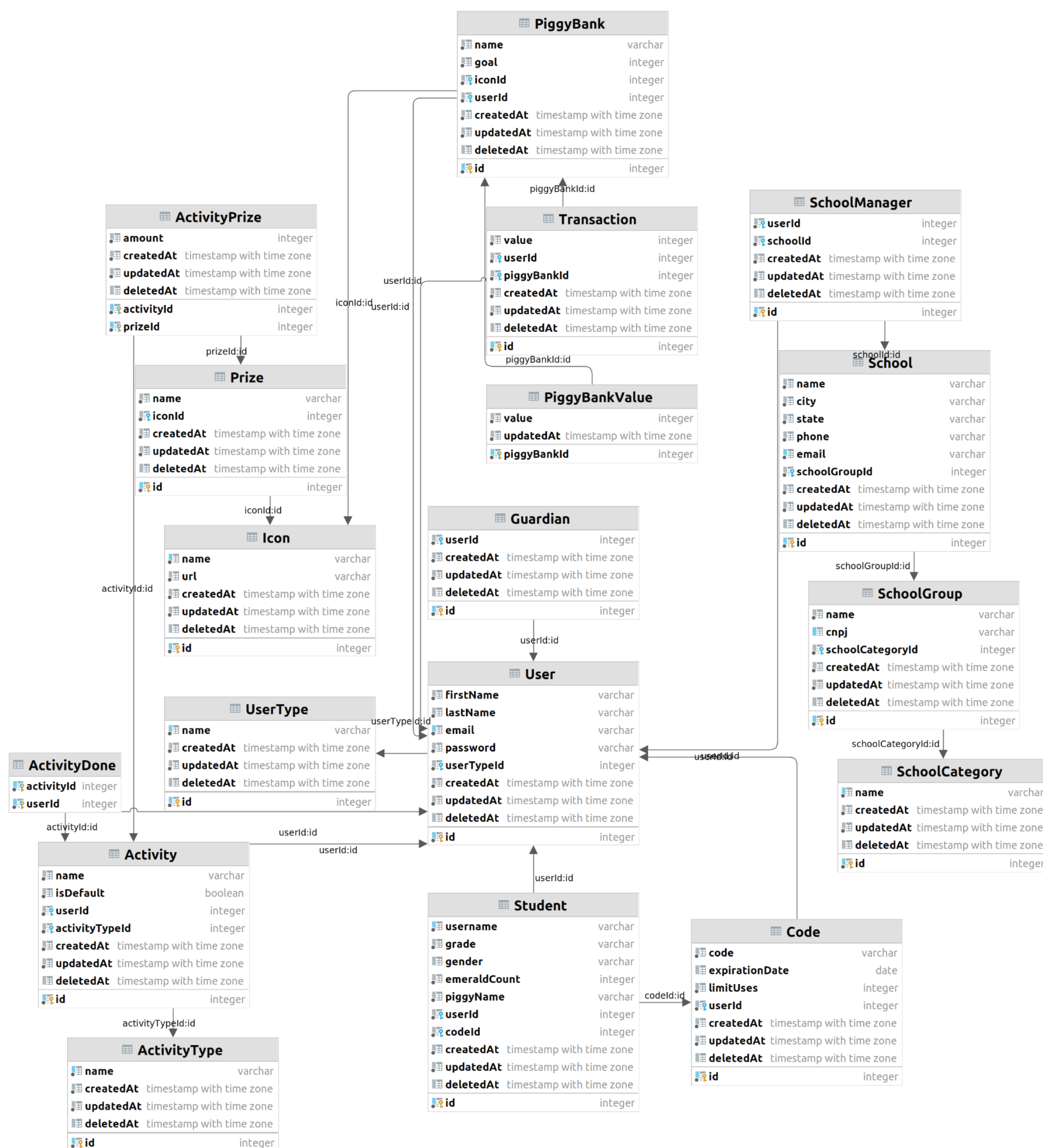
O objetivo desse tópico é discutir e explicar sobre a necessidade de uma abordagem híbrida e as estruturas criadas para o consumo da API do Cofrinho, seja no banco PostgreSQL, explicando a ideia de cada uma das tabelas e seus relacionamentos, seja no banco MongoDB, explicando sobre a estrutura esperada e o validador definido.

O uso de um banco de dados de documentos como o MongoDB foi abordado após a etapa inicial de modelagem, uma vez que as tabelas relacionadas ao módulo educacional estavam muito extensas e repletas de relacionamentos complexos. Além disso, as potenciais diferenças no modelo de cada tipo de atividade deixavam a implementação cada vez mais lenta e de difícil explicação. Surgiu então a ideia de utilizar um banco de documentos, que pode armazenar um objeto JSON muito complexo a um preço baixíssimo, como mostram os resultados de Quillici e Schiavon (2021) e Knijnik (2022). Uma vez que os dados contidos em um objeto não precisam se relacionar com outros (uma atividade e seus materiais não podem ser parte de outra atividade), esta abordagem se mostrou plausível e foi adicionada no plano de desenvolvimento.

A modelagem realizada na figura 4 retrata o modelo SQL completo utilizado na aplicação. Como ele é muito extenso e possui muitos relacionamentos, sua observação pode ser muito complicada. Assim, optou-se por separar sua visualização em quatro módulos: Usuário, Escola, Cofrinho e Educacional. Eles serão apresentados e explicados em seguida.

A maioria das tabelas contam com alguns atributos em comum, sendo id o identificador único de cada registro, createdAt, updatedAt e deletedAt campos para definir as datas de criação, atualização e deleção, respectivamente. São utilizados como forma de manter a integridade dos dados e prover uma forma de geração de relatórios históricos. Vale ressaltar que, por conta disso, os registros são *soft-deleted* ou fracamente deletados, essencialmente marcados como deletados mas não removidos do banco de dados. Isso se dá pela necessidade de manter históricos para gerar relatórios ou analisar o comportamento dos usuários.

Figura 4 - Modelo do banco de dados completo.



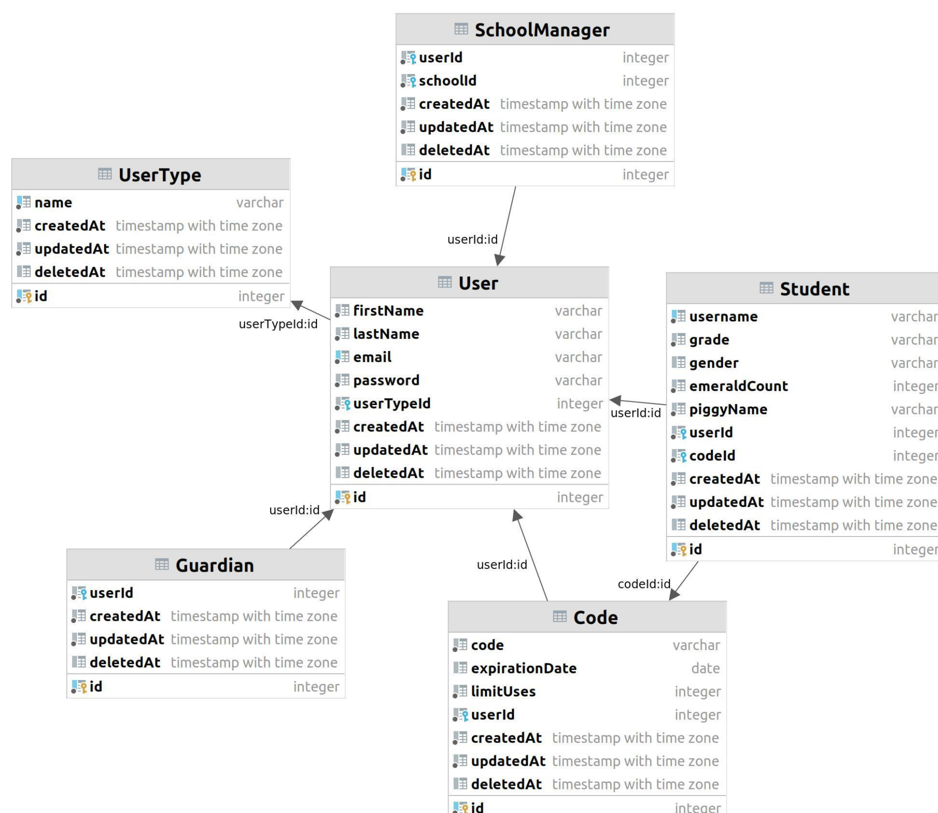
Fonte: Do autor, 2023.

4.8.1 MÓDULO DE USUÁRIOS

Composto pelas tabelas: User, UserType, Student, SchoolManager e Code, como indica a figura 5. Este módulo conta com todos os dados relacionados a um usuário. Enquanto a tabela UserType armazena valores redundantes, já que é possível descobrir se um usuário é um “SchoolManager”, por exemplo, ao encontrar seu ID nessa tabela, essa foi a alternativa mais barata computacionalmente que a equipe encontrou. Sua falha estrutural (onde um usuário pode constar como SchoolManager e Student simultaneamente), é tratada apenas na regra de negócio da aplicação. O uso de códigos para acesso dos alunos visa uma implementação futura ao MVP, onde os códigos serão comercializados.

A tabela Code conta ainda com o atributo userId, que no contexto do MVP poderia ser substituído por uma referência a escola. Porém, no futuro, usuários de outros tipos poderão também gerar códigos, logo essa implementação faz mais sentido. Vale ressaltar que, apesar de possível por meio do banco, há uma validação no código do sistema que impede que usuários do tipo "Aluno" criem códigos.

Figura 5 - Modelo do módulo de usuários.

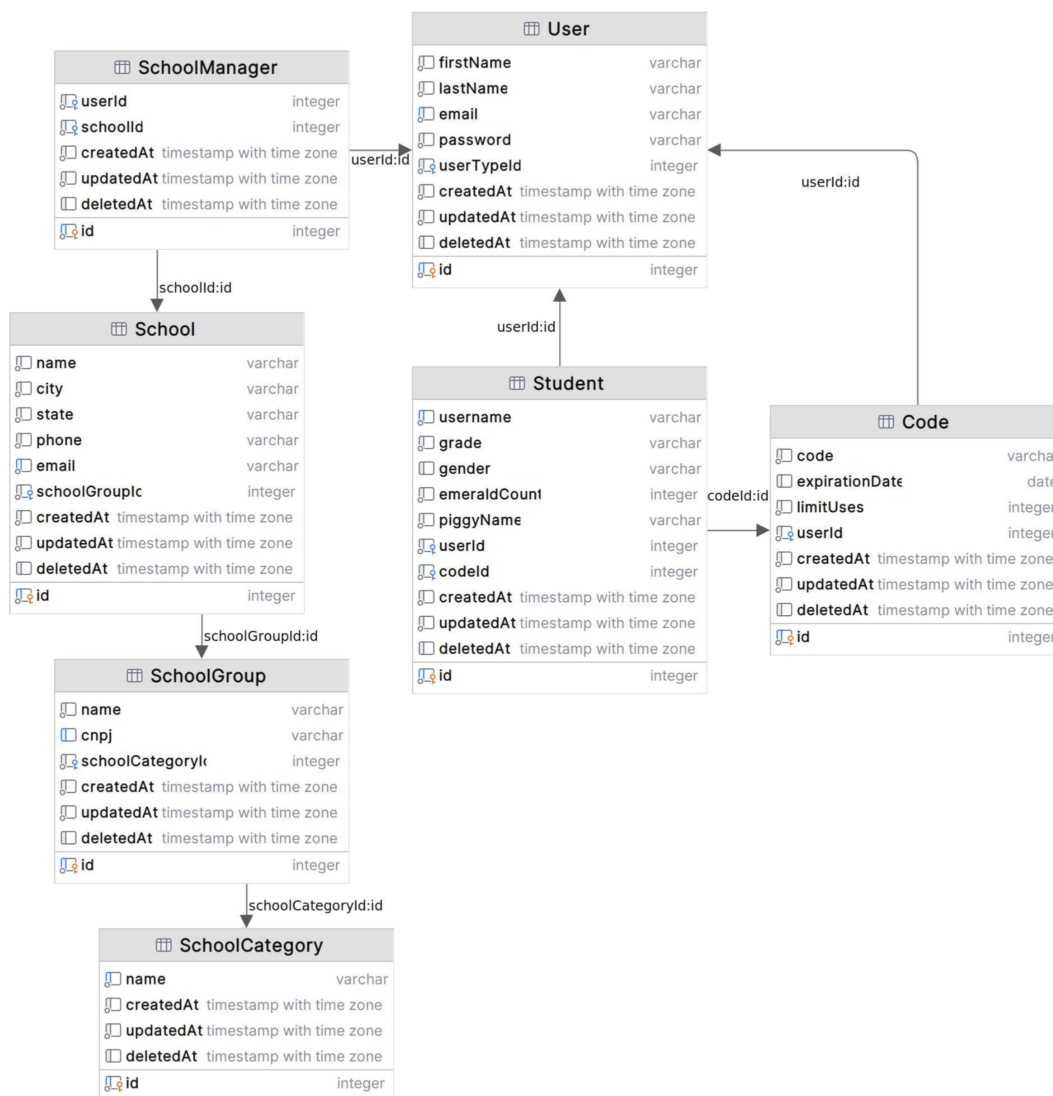


Fonte: Do autor, 2023.

4.8.2 MÓDULO DE ESCOLAS

A figura 6 estampa o módulo de escola, que apresenta dados relacionados a essa entidade no sistema. É composto pelas tabelas School, SchoolCategory, SchoolGroup, SchoolManager, User, Code e Student. Não houveram grandes decisões relacionadas a esse módulo. Um detalhe importante é sobre a criação dos usuários do tipo "Escola", que demandam mais de uma operação (inserção na tabela User e SchoolManager). Essa dualidade é controlada em código, pelo sistema. School Category e Group representam as categorias e grupos de escolas na plataforma. As categorias as separam, inicialmente, como públicas ou privadas, enquanto os grupos segregam as públicas como federais, estaduais ou municipais e as privadas quanto a seu grupo educacional.

Figura 6 - Modelo do módulo de escolas.



Fonte: Do autor, 2023.

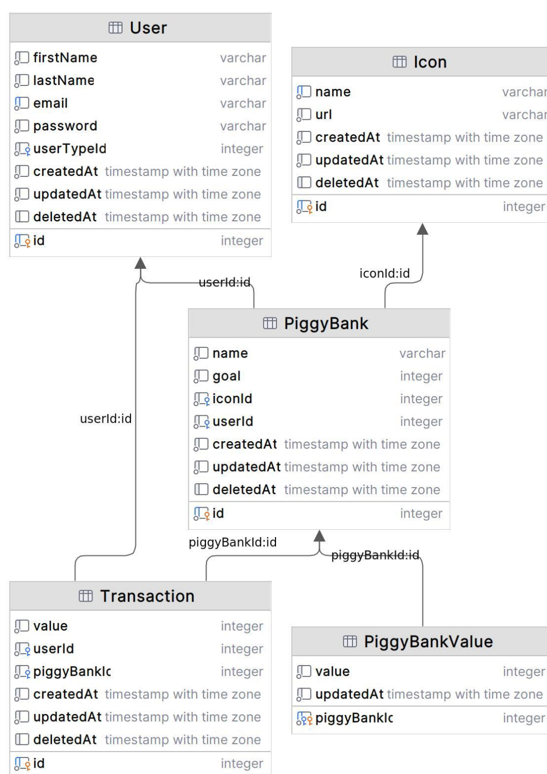
4.8.3 MÓDULO DE COFRINHOS

Como consta na figura 7, o módulo é composto pelas tabelas Icon, PiggyBank, PiggyBankValue, Transaction e User. Todos os dados relacionados aos cofrinhos dos usuários são descritos neste tópico. A decisão mais difícil em relação a este módulo foi a de como armazenar e exibir o valor acumulado dos cofrinhos. Visto que armazenar as transações eh um requisito da aplicação, as alternativas eram somar todas as transações a cada vez que o valor total era requerido, seja por meio de uma query convencional ou por meio de Views ou Views Materializadas, ou manter o valor total somado no cofrinho.

A primeira alternativa tem um problema sério de performance, visto que demanda muitos cálculos, que tendem a ficar cada vez mais longos. A segunda, porém, tem um

problema de confiabilidade, visto que ambos valor total e soma das transações devem sempre apresentar o mesmo valor, demandando duas operações por inserção e travando a tabela com frequência. Assim, foi escolhida uma abordagem mista, onde criou-se a tabela PiggyBankValue, que armazena uma soma parcial do valor do cofrinho até determinada data. As transações dessa data em diante devem ser somadas para obter o valor total. Assim, há uma rotina definida em código, para atualizar o valor dessa tabela com alguma frequência.

Figura 7 - Modelo do módulo de cofrinhos.

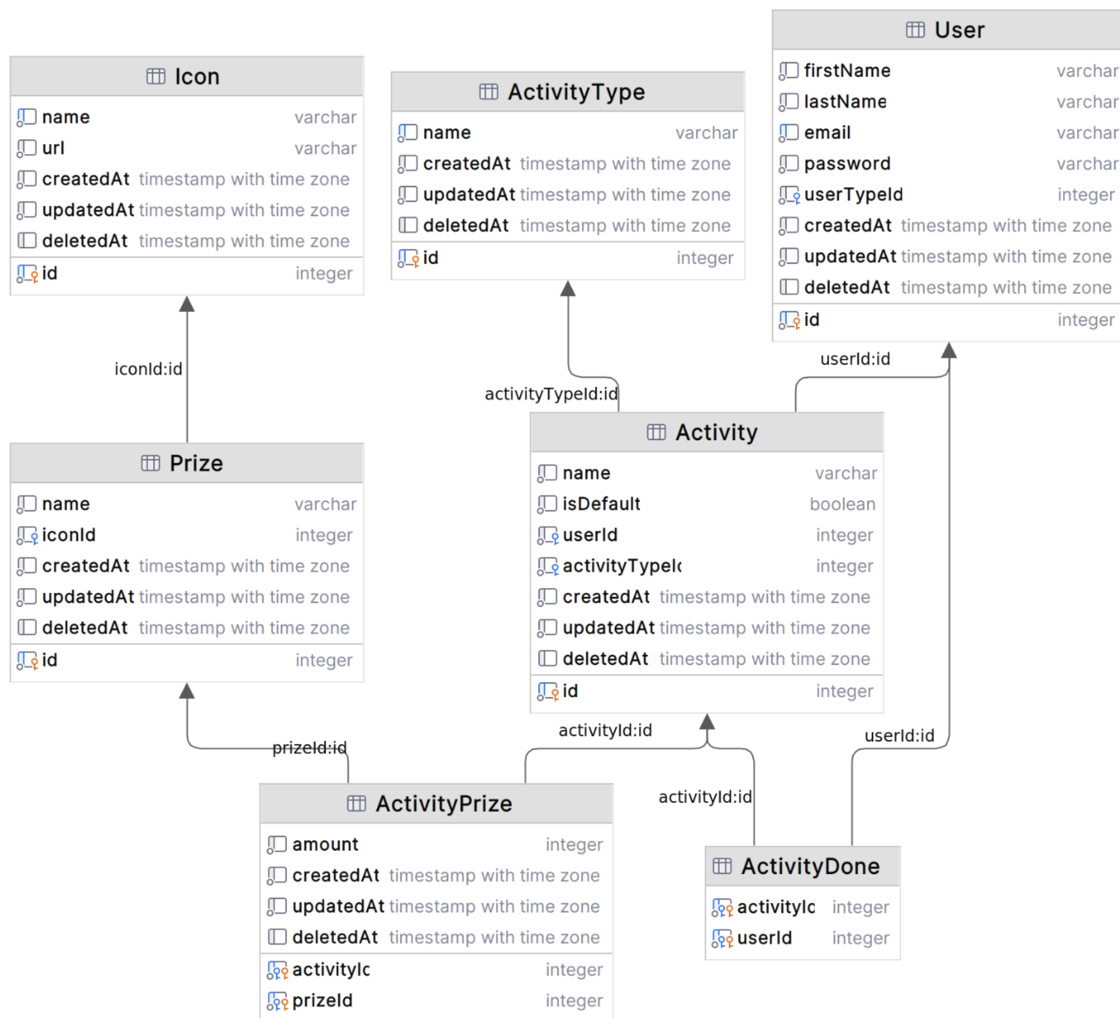


Fonte: Do autor, 2023.

4.8.4 MÓDULO EDUCACIONAL

Evidenciado na figura 8, o módulo é composto pelas tabelas Icon, ActivityType, Activity, Prize, Activity e ActivityDone. Icon diz respeito ao ícone apresentado por uma premiação. ActivityType descreve os tipos de atividades que podem existir (no banco MongoDB, seu nome é utilizado como nome da coleção). A Activity carrega os dados de uma atividade, mostrando seu nome, tipo, se é padrão, ou seja, todos os usuários têm acesso. Seu id é utilizado como chave para buscá-lo no banco de documentos. Prize e ActivityPrize, respectivamente, representam os prêmios que existem no sistema e como eles se relacionam com as atividades. O User, nesse contexto, é utilizado para armazenar quem criou uma atividade e quem a realizou, sendo este último identificado na tabela ActivityDone.

Figura 8 - Modelo do módulo educacional.



Fonte: Do autor, 2023.

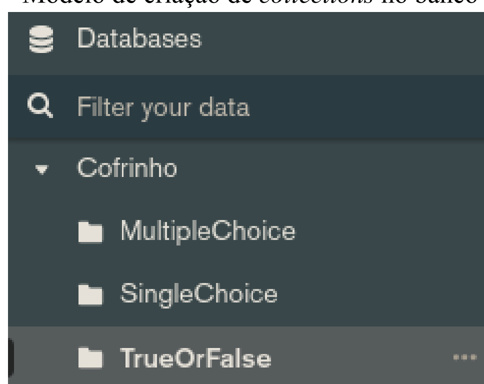
A figura 9 mostra um exemplo de validador do MongoDB, evidenciando o modelo utilizado para validar o esquema de uma atividade do tipo *TrueOrFalse*. A figura 10, evidencia como ficou o banco de dados do Cofrinho no MongoDB e suas coleções. Vale ressaltar que o modelo escolhido deve passar por alterações nas fases seguintes de implementação, onde haverá mais tempo hábil para uma modelagem mais adequada.

Figura 9 - Esquema de validação da *collection* relativa às atividades *TrueOrFalse.*

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'TrueOrFalseSchemaValidator',
    required: [
      'id',
      'title',
      'question',
      'availableAnswers',
      'correctAnswer'
    ],
    properties: {
      id: {
        bsonType: 'number',
        title: 'Id Reference For Activity',
        description: '"id" must be a number and is required!'
      },
      title: {
        bsonType: 'string',
        title: 'Activity Title',
        description: '"title" must be a string and is required!'
      },
      question: {
        bsonType: 'string',
        title: 'Activity Question',
        description: '"question" must be a string and is required!'
      },
      availableAnswers: {
        bsonType: 'array',
        title: 'Activity Available Answers',
        description: '"availableAnswers" must be an array of string and is required!',
        uniqueItems: true,
        items: {
          bsonType: 'string'
        }
      },
      correctAnswer: {
        bsonType: 'number',
        title: 'Activity Correct Answer Index',
        description: '"correctAnswer" must be a index number and is required!'
      }
    }
  }
}
```

Fonte: Do autor, 2023.

Figura 10 - Modelo de criação de *collections* no banco MongoDB.



Fonte: Do autor, 2023.

4.9 DESENVOLVIMENTO DA API

Essa fase foi, tecnicamente, a mais longa envolvida no processo de desenvolvimento, visto que contou com a criação de quase toda a base e código da API do Cofrinho. Porém, seu

processo foi bem repetitivo e cabível de poucos relatos relevantes. De forma geral foi pautado nos seguintes processos:

- Planejamento: quebrar cada tarefa em atividades menores e dar um prazo para cada uma, de forma a cumprir seu cronograma;
- Migrações: caso necessário, migrações eram desenvolvidas para alterar o banco conforme necessário;
- Entidades: caso necessário, as classes de entidades envolvidas eram criadas ou alteradas, conforme as alterações no banco;
- Repositório: caso necessário, os repositórios SQL eram criados ou alterados, conforme as alterações no banco e necessidades dos serviços;
- DTOs (*Data Transfer Object*, objeto de transferência de dados em tradução livre): caso necessário, os DTOs de entrada (modelos que definem os dados a serem enviados pelo usuário) eram criados ou alterados;
- Validadores: caso necessário, os validadores de domínio do serviço eram criados ou alterados;
- Serviços: caso necessário, os serviços e suas regras de negócio eram criados ou alterados;
- Controlador: caso necessário, rotas e parâmetros envolvidos eram criados ou alterados;
- Testes: os testes (mapeados pelo PO e testadores) requisitados na funcionalidade foram criados ou alterados.

Alguns dos incidentes ocorridos foram causados por falhas no modelo de permissionamento, onde usuários poderiam burlar alguma verificação e resgatar itens que não deveriam poder ver. Esse problema foi resolvido com a criação de um middleware que injeta as regras de permissionamento, e uma função que utiliza esses valores para bloquear recursos de serem acessados no banco de dados, como consta na tarefa #20 da tabela 6. Nenhuma biblioteca foi utilizada para este processo.

Outro problema identificado foi no cumprimento dos prazos. O cadastro de atividades, por conta de suas entradas variáveis, demandou uma segunda etapa de planejamento, ainda maior que as convencionais. Outras tarefas, como #9, #10, #17 e #18 tiveram o mesmo problema, demandando que fossem finalizados apenas no ciclo seguinte. Algumas outras tarefas sofreram da situação oposta, onde foram completadas antes do prazo, como #8, #13 e #16, que demandaram por adiantamento de tarefas futuras.

5 RESULTADOS

Ao final da etapa de desenvolvimento obteve-se uma API HTTP RESTful completamente funcional, capaz de satisfazer os requisitos propostos neste relatório. Esta seção tem como objetivo apresentar os resultados obtidos com a execução do trabalho, detalhando as funcionalidades desenvolvidas, bem como as entidades relacionadas.

5.1 ENTIDADES

As entidades criadas, bem como suas características, constam na tabela 7. Cada registro representa uma entidade de resposta, ou seja, os dados que serão enviados ao consumir a API, apresentando também seus atributos e a tipagem de cada um deles.

Tabela 7 - Tabela de entidades.

Entidade	Atributo	Tipo	Obrigatório
UserType	id	numeral	Sim
	name	textual	Sim
	users	User[]	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
User	id	numeral	Sim
	firstName	textual	Sim
	lastName	textual	Sim
	email	textual	Sim
	userType	UserType	Sim
	codes	Code[]	Sim
	guardian	Guardian	Não
	schoolManager	SchoolManager	Não
	student	Student	Não
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não

Entidade	Atributo	Tipo	Obrigatório
Code	id	numeral	Sim
	code	textual	Sim
	expirationDate	data	Sim
	user	User	Sim
	students	Student[]	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
Guardian	id	numeral	Sim
	user	User	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
SchoolManager	id	numeral	Sim
	user	User	Sim
	school	School	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
Student	id	numeral	Sim
	username	textual	Sim
	grade	textual	Sim
	emeraldCount	numeral	Sim
	piggyName	textual	Sim
	user	User	Sim
	code	Code	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não

Entidade	Atributo	Tipo	Obrigatório
School	id	numeral	Sim
	name	textual	Sim
	city	textual	Sim
	state	textual	Sim
	phone	textual	Sim
	email	textual	Sim
	schoolGroup	SchoolGroup	Sim
	schoolManagers	SchoolManager[]	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
SchoolGroup	id	numeral	Sim
	name	textual	Sim
	cnpj	textual	Sim
	schoolCategory	SchoolCategory	Sim
	schools	School[]	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
SchoolCategory	id	numeral	Sim
	name	textual	Sim
	schoolGroups	SchoolGroup[]	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não

Entidade	Atributo	Tipo	Obrigatório
PiggyBank	id	numeral	Sim
	name	textual	Sim
	goal	numeral	Sim
	icon	Icon	Sim
	user	User	Sim
	value	numeral	Sim
	transactions	Transaction[]	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
Icon	id	numeral	Sim
	name	textual	Sim
	url	textual	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
Transaction	id	numeral	Sim
	value	numeral	Sim
	user	User	Sim
	piggyBank	PiggyBank	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não

Entidade	Atributo	Tipo	Obrigatório
Activity	id	numeral	Sim
	name	textual	Sim
	isDefault	booleano	Sim
	user	User	Sim
	activityType	ActivityType	Sim
	activityPrize	ActivityPrize	Sim
	activity	JSON	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
ActivityType	id	numeral	Sim
	name	textual	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
ActivityPrize	id	numeral	Sim
	amount	numeral	Sim
	activity	Activity	Sim
	prize	Prize	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não
Prize	id	numeral	Sim
	name	textual	Sim
	icon	Icon	Sim
	createdAt	data	Sim
	updatedAt	data	Sim
	deletedAt	data	Não

Fonte: Do autor, 2023.

5.2 ROTAS DA API

O resultado final da API foi sintetizado por meio da tabela 8, que mostra quais foram as rotas HTTP disponíveis, como acessá-las e quais suas respostas. A coluna nome identifica a rota com seu objetivo, método indica qual verbo HTTP foi selecionado, caminho mostra o caminho completo para atingir a rota (aqueles que possuem ‘:’ são considerados variáveis), resposta diz qual será a resposta em caso de sucesso, utilizando as entidades definidas na tabela 7, e logado indica se a rota deve ser acessada com autenticação ou não.

Tabela 8 - Rotas HTTP.

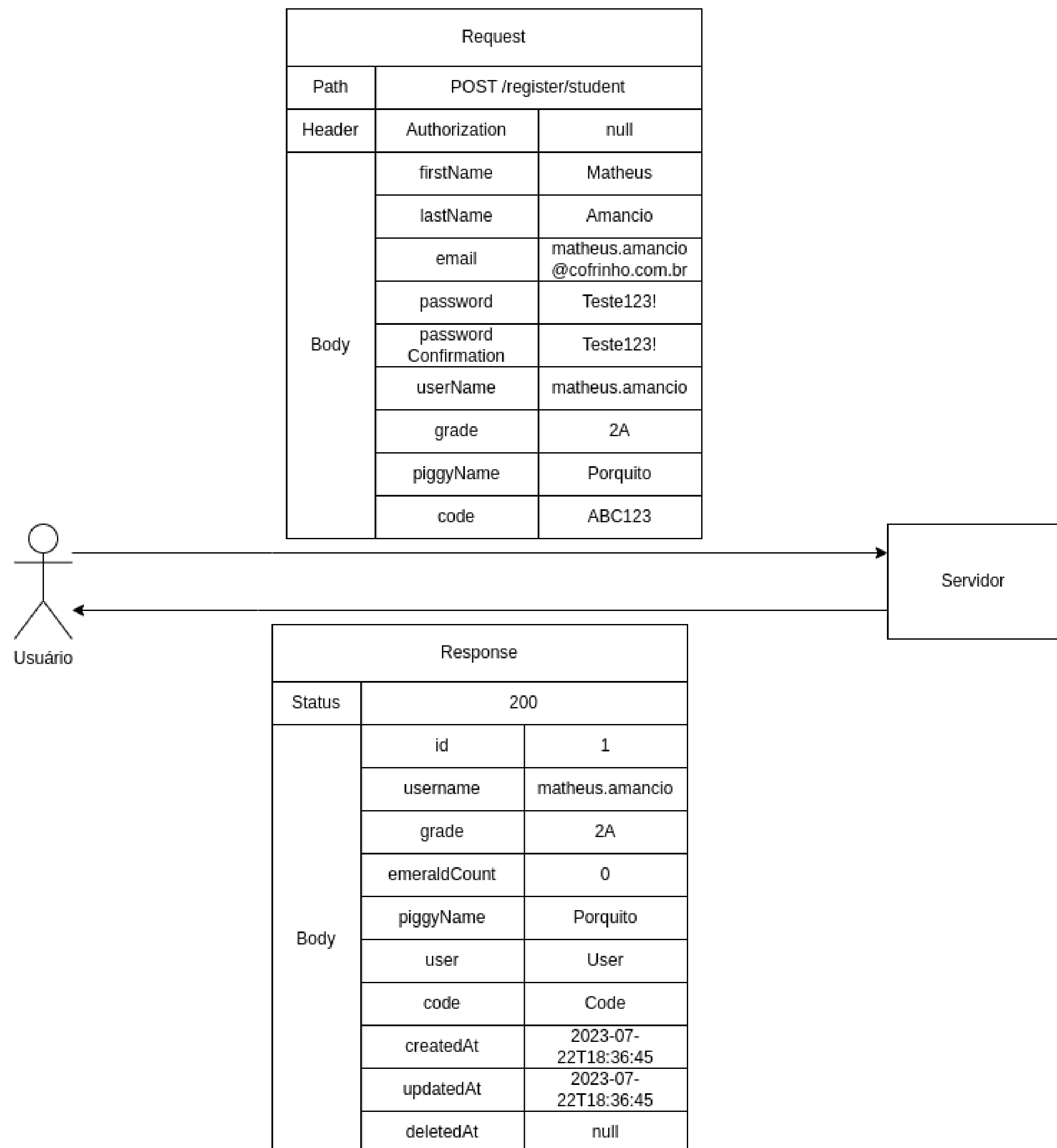
Nome	Método	Caminho	Resposta	Logado
Register School Manager	POST	/register/schoolManager	SchoolManager	Não
Register Guardian	POST	/register/guardian	Guardian	Não
Register Student	POST	/register/student	Student	Não
Login	POST	/login	User	Não
Upgrade to Master	PATCH	/admin/upgrade	User	Sim
Create Code	POST	/codes	Code	Sim
Get Code	GET	/codes/:idCode	Code	Sim
List Codes	GET	/codes	Code[]	Sim
Create School Category	POST	/schoolCategories	SchoolCategory	Sim
Get School Category	GET	/schoolCategories/:idSchoolCategory	SchoolCategory	Sim
List School Categories	GET	/schoolCategories	SchoolCategory[]	Sim
Create School Group	POST	/schoolCategories/:idSchoolCategory/schoolGroups	SchoolGroup	Sim
Get School Group	GET	/schoolCategories/:idSchoolCategory/schoolGroups/:idSchool Group	SchoolGroup	Sim
List School Groups	GET	/schoolCategories/:idSchoolCategory/schoolGroups	SchoolGroup[]	Sim
Create School	POST	/schools	School	Sim
Get School	GET	/schools/:idSchool	School	Sim

List Schools	GET	/schools	School[]	Sim
Create Piggy Bank	POST	/piggyBanks	PiggyBank	Sim
Get Piggy Bank	GET	/piggyBanks/:idPiggyBank	PiggyBank	Sim
List Piggy Banks	GET	/piggyBanks	PiggyBank[]	Sim
Update Piggy Bank	PATCH	/piggyBanks/:idPiggyBank	PiggyBank	Sim
Delete Piggy Bank	DELETE	/piggyBanks/:idPiggyBank	PiggyBank	Sim
Create Transaction	POST	/piggyBanks/:idPiggyBank/transactions	Transaction	Sim
Get Transaction	GET	/piggyBanks/:idPiggyBank/transactions/:idTransaction	Transaction	Sim
List Transactions	GET	/piggyBanks/:idPiggyBank/transactions	Transaction[]	Sim
Create Activity	POST	/activities	Activity	Sim
Get Activity	GET	/activities/:idActivity	Activity	Sim
List Activities	GET	/activities	Activity[]	Sim
Assign Activity Prize	POST	/activities/:idActivity/prize	ActivityPrize	Sim
Get Activity Prize	GET	/activities/:idActivity/prize	ActivityPrize	Sim
Delete ActivityPriz e	DELETE	/activities/:idActivity/prize	ActivityPrize	Sim
Recommend Activity	GET	/activities/recommended	Activity	Sim
Assign Activity Done	POST	/activities/:idActivity/resolve	ActivityDone	Sim
Get Activity Done	GET	/activities/:idActivity/resolve	ActivityDone	Sim
List Activities Done	GET	/activities/resolve	ActivityDone[]	Sim

Fonte: Do autor, 2023.

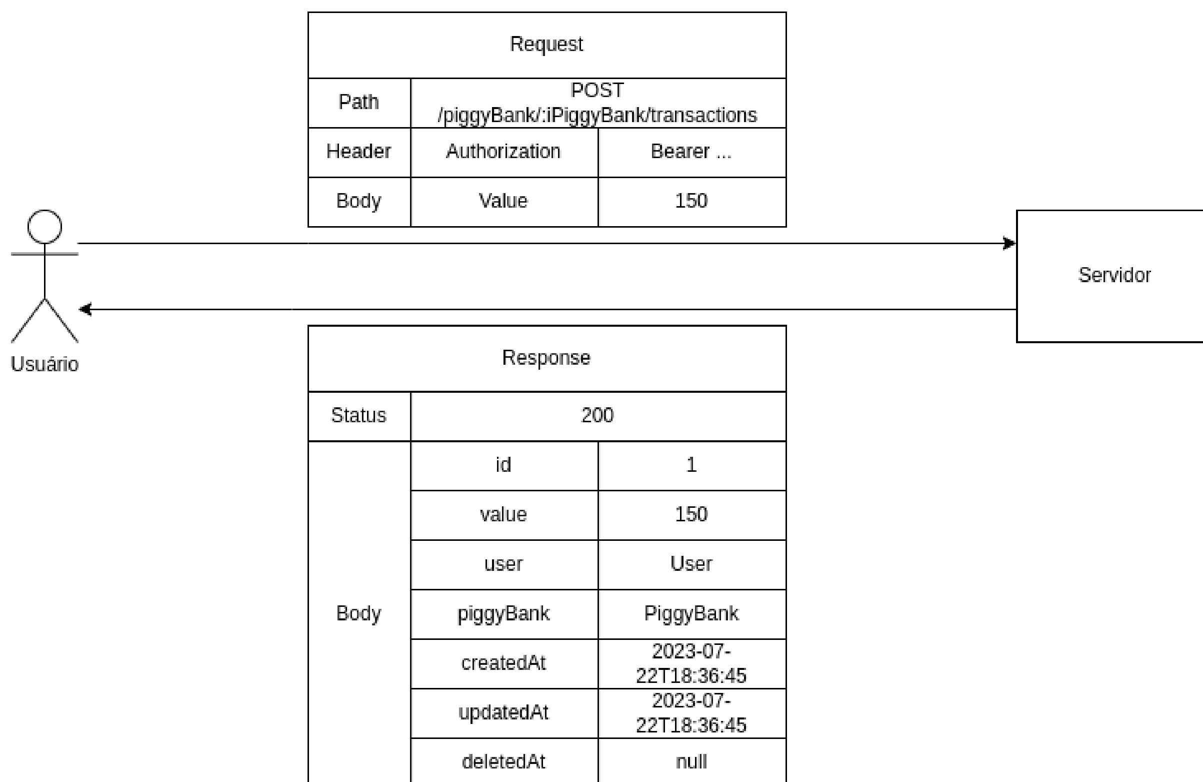
As figuras 12 e 13, por sua vez, detalham melhor o funcionamento e fluxo de dados de duas requisições, retratando situações com o usuário logado e não logado. As rotas escolhidas foram de cadastro de estudante e criação de transações.

Figura 11 - Cadastro de estudante.



Fonte: Do autor, 2023.

Figura 12 - Registro de Transações.



Fonte: Do autor, 2023.

A API conta também com padronização de respostas, isto é, segue alguns critérios para definir as respostas enviadas aos usuários. Respostas de erro, por exemplo, são enviadas conforme critérios definidos na aplicação. A tabela 9 mostra o status HTTP enviado em cada situação, bem como um texto padrão para casos de erro.

Tabela 9 - Padrão de respostas.

Nome	Status	Mensagem Padrão
Ok	200	-
Erro genérico do cliente	400	Houve um erro na requisição
Não autenticado	401	Usuário não autenticado
Não autorizado	403	Usuario nao tem permissao
Duplicado	409	Registro duplicado
Não encontrado	404	Registro não encontrado
Erro de validação	422	Houve um erro de validação
Erro genérico da aplicação	500	Houve um erro ao processar sua requisição. Tente novamente mais tarde

Fonte: Do autor, 2023.

6 CONCLUSÃO

O projeto Cofrinho busca levar educação financeira para jovens de forma lúdica e simples. O funcionamento de seu sistema depende de diferentes ferramentas em paralelo, como seu aplicativo, API e bases de dados, e o processo de construção dos mesmos foi muito auxiliado pela documentação e planejamento requeridos por este trabalho.

Ele foi dividido em etapas, que envolveram o mapeamento de requisitos, planejamento de gestão de desenvolvimento, desenho de uma arquitetura, tanto do sistema como um todo, como do código construído, seleção de ferramentas e desenvolvimento de um software baseado em métodos robustos.

Este trabalho propõe e relata o desenvolvimento de uma API RESTful para cumprir com os requisitos de uma startup nascente do ramo de EdTechs, expondo toda a metodologia seguida e os acertos e erros envolvidos por elas. Contém também uma breve etapa de tratamento para adequar-se à LGPD, um importante passo para todas as empresas de TI no Brasil.

O MVP desenvolvido nesta etapa ainda será colocado para testes com usuários reais, logo, não é possível validar o trabalho como um todo, em especial as questões que envolvem a estabilidade da aplicação com um volume de tráfego maior. Ainda assim, pode-se dizer que o desenvolvimento ocorreu conforme o esperado, e atendeu a todas as expectativas e demandas propostas a ele. Dentre o plano original que ditou a elaboração deste projeto, a única fase que não pode ser detalhada neste documento foi a de implantação na nuvem, visto que faltam recursos financeiros para tal.

Como descrito por todo o projeto, foram necessárias adaptações nas metodologias escolhidas para o desenvolvimento, como por exemplo as mudanças nos papéis Scrum e as adaptações nos padrões CLEAN e SOLID. Isso se deu por sua complexidade de implementação, que, em equipes pequenas, pode atrapalhar mais do que ajudar. Ainda assim, suas escolhas foram assertivas, pois não só colocaram o projeto num patamar similar ao da indústria de software moderna, como ajudaram o autor a entender a fundo cada uma delas, ao ponto de propor alterações para a realidade enfrentada.

As disciplinas cursadas durante a graduação se mostraram importantes para a elaboração do trabalho como um todo. Mas não por cobrirem todas as áreas abordadas profundamente, ou por ensinarem determinada tecnologia, e sim por darem uma base teórica que auxilia no entendimento aprofundado de técnicas e tecnologias, e que ajudaram o autor a tomar decisões e dar respostas bem estruturadas. As disciplinas de banco de dados, ou de arquitetura distribuída são um exemplo claro disto. Outras, como arquitetura de

computadores, por exemplo, são menos claras, mas dão uma base sólida em conhecimentos da área de tecnologia da informação. Um projeto de extensão e um estágio, por outro lado, auxiliaram em questões mais técnicas e agilizaram alguns processos de pesquisa, visto a experiência que agregam. Uma boa relação entre ambos os conhecimentos foram ótimos para o desenvolvimento do trabalho.

Na concepção inicial da API proposta para o Cofrinho, não haviam tantos empecilhos e decisões difíceis como as que se tornaram realidade durante as fases de planejamento, cobertas pelo relatório, evidenciando que poderia haver muita dificuldade durante o desenvolvimento caso esta etapa não fosse realizada. No contexto de uma empresa, isso afeta prazos, qualidade e até mesmo dinheiro. A elaboração do relatório, mesmo que tenha aumentado a burocracia envolvida no processo de desenvolvimento, se mostrou uma alternativa adequada, repleta de possibilidades não encontradas caso ela fosse ignorada.

REFERÊNCIAS

AMBLER, S. W. Mapping objects to relational databases: **O/R mapping in detail.** Ambysoft Inc., 2003.
ASHLEY, K.; CARVALHO, M.; LOPES, R. Análise de desempenho de estratégias de autoscaling vertical e horizontal: **um estudo de caso com o Kubernetes.** Anais Estendidos do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC). SBC, 23 de maio de 2022. Disponível em: <https://sol.sbc.org.br/index.php/sbrc_estendido/article/view/21437>. Acesso em: mai. 2023.
AMAZON WEB SERVICES. AWS: **Benefícios**. AWS.amazon.com, [s. l.], c2023 Disponível em: <https://aws.amazon.com/pt/application-hosting/benefits/>. Acesso em: abr. 2023.
AMAZON WEB SERVICES. AWS: **O que é API RESTful?**. AWS.amazon.com, [s. l.], c2023 Disponível em: <https://aws.amazon.com/pt/what-is/restful-api/>. Acesso em: jul. 2023.
BARSOTI, N.; GIBERTONI, D. **Impacto que o Sequelize traz para o desenvolvimento de uma API construída em Node. JS com Express, JS.** Revista Interface Tecnológica, v. 17, n. 2, p. 231–243, 18 dez, 2020.
BRASIL. Lei nº 13.709, de 14 de agosto de 2018. **Lei Geral de Proteção de Dados Pessoais (LGPD)**. Brasília, DF: Presidência da República, 2018. Disponível em: https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/L13709.htm. Acesso em: jun. 2022.
CALEGARI, G. **Arquiteturas de Sistemas Distribuídos.** Calegari.dev, [s. l.], 20 jan. 2021. Disponível em: <https://calegari.dev/posts/arquiteturas-de-sistemas-distribuidos/>. Acesso em: abr. 2023.
BRASIL. Ministério da Educação. **Base Nacional Comum Curricular.** Brasília, 2018.
CANONICAL. Ubuntu: **Basic installation.** Ubuntu.com, [s. l], [2023]. Disponível em: <https://ubuntu.com/server/docs/installation>. Acesso em: dez. 2022.
CARNEIRO, R. A. et al. **O valor percebido pelo cliente na gestão de projetos de software, utilizando o framework scrum.** Gestão e Projetos: GeP, v. 13, n. 3, p. 149–176, 2022.
CAVALCANTE, Damares da Silva. **Utilização de desenvolvimento centrado na arquitetura para a construção de sistemas de informação complexos**. Trabalho de conclusão de curso - UFAL Campus Arapiraca. 2019.
CNC (Confederação Nacional do Comércio de Bens, Serviços e Turismo). **Pesquisa de Endividamento e Inadimplência do Consumidor** (Peic), 2022.
CONFEDERAÇÃO NACIONAL DE DIRIGENTES LOJISTAS - CNDL Brasil. **Inadimplência bate recorde e atinge 66 milhões de consumidores, aponta CNDL/SPC Brasil.** Site.cndl.org.br, Brasília, 18 abr. 2023. Disponível em: <https://site.cndl.org.br/inadimplencia-bate-recorde-e-atinge-66-milhoes-de-consumidores-aponta-cndlspc-brasil/>. Acesso em: jun. 2023.
CLOUDFLARE. Cloudflare: **O que é HTTP?**. Cloudflare.com, [s. l.], c2023 Disponível em: <https://www.cloudflare.com/pt-br/learning/ddos/glossary/hypertext-transfer-protocol-http/>. Acesso em: jul. 2023.

COOPER, R.; VLADECK, D. The Lean Product Playbook: **How to Innovate with Minimum Viable Products and Rapid Customer Feedback.** John Wiley & Sons, 2016.
COUTO, H. et al. **Uma Abordagem Experimental para Avaliar o Desempenho do Banco de Dados Open-Source PostgreSQL.** Anais da Escola Regional de Informática de Goiás (ERI-GO). Anais da X Escola Regional de Informática de Goiás. SBC, 25 out. 2022. Disponível em: <https://sol.sbc.org.br/index.php/erigo/article/view/22533>. Acesso em: fev. 2023.
DOCKER. Docker Docs: **how to build, share, and run applications.** Docs.docker.com, [s. l.], c2023. Disponível em: <(https://docs.docker.com/)> Acesso em: mai. 2023.
ESTRATÉGIA NACIONAL DE EDUCAÇÃO FINANCEIRA - ENEF. **Conceito de Educação Financeira no Brasil.** Vidaedinheiro.gov.br, Brasil, 2017. Disponível em: <https://www.vidaedinheiro.gov.br/educacao-financeira-no-brasil> Acesso em: jun. 2023.
FARIA, Afro Netto Nunes. **O uso das técnicas de levantamento de requisitos no mercado atual.** 63 f. Trabalho de Conclusão de Curso (Especialização em Gestão de Tecnologia da Informação e Comunicação) - Universidade Tecnológica Federal do Paraná, Curitiba, 2016.
FELIPETTO, N.; BASSO, F. P. **Em Direção ao Desenvolvimento Dirigido por Modelos de Sistemas Web Baseados em Scripts.** Anais do Salão Internacional de Ensino, Pesquisa e Extensão, v. 12, n. 2, 4 dez. 2020.
FIESLER, C. et al. Minimum Viable Research: **A Guide to Early-Stage User Research for Technology Development Teams.** In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. ACM, 2018.
GITHUB, OCTOVERSE. **The top programming languages**. 2022. Disponível em <https://octoverse.github.com/2022/top-programming-languages>. Acesso em Jun, 2023.
GOMES, R. L.; WILLRICH, R.; JESÚS HERNÁNDEZ PALANCO, G. **Arquiteturas de Distribuição. In: Sistemas Colaborativos.** Rio de Janeiro: Elsevier Brasil, 2019.
INTERNATIONAL BUSINESS MACHINES CORPORATION - IBM. **O que é arquitetura de três camadas (tiers).** Ibm.com, [s. l.], [2023?]. Disponível em: <https://www.ibm.com/br-pt/topics/three-tier-architecture>. Acesso em: mai. 2023.
INTERNATIONAL BUSINESS MACHINES CORPORATION - IBM. **O que é uma API de REST?.** Ibm.com, [s. l.], [c2023]. Disponível em: <https://www.ibm.com/br-pt/topics/rest-apis>. Acesso em: jul. 2023.
KNIJNIK, D. M. **Comparação de SGBDs mongoDB e postgreSQL para jogos digitais,** 2022.
LAGE JUNIOR, M.; GODINHO FILHO, M. Adaptações ao sistema kanban: **revisão, classificação, análise e avaliação.** Gestão & Produção, v. 15, n. 1, p. 15-28, 2008.
MARQUES, Márcia Soares. **Metodologia Agiliza.** 126f. Dissertação (Mestrado Profissional em Ciência, Tecnologia e Inovação) - Escola de Ciências e Tecnologia, Universidade Federal do Rio Grande do Norte, Natal, 2021.
MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. 1ª ed. Pearson, 2017.
MARTIN, Robert C. **Código Limpo: Habilidades Práticas do Agile Software**. 1ª ed. Atlas Book, 2009.

MARTIN, Robert C. **Design Principles and Design Patterns**. 2000. Disponível em <http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf>. Acesso em: mar. 2023.
MARTINS, F. et al. Kubernetes: **uma plataforma para orquestração e gestão de clusters.** In: WORKSHOP DE COMPUTAÇÃO EM NUVENS E APLICAÇÕES (WCNA), v. 10., Belém. Anais […]. Porto Alegre: Sociedade Brasileira de Computação, 2019.
MICROSOFT. Typescript: **The starting point for learning TypeScript.** Typescriptlang.org, [s. l.], c2023. Disponível em: <https://www.typescriptlang.org/docs/>. Acesso em: jan. 2023.
MONGODB. **MongoDB Documentation.** Mongodb.com, [s. l.], c2023. Disponível em: <(https://www.mongodb.com/docs/)>. Acesso em: mai 2023.
MONGODB. **Pricing | MongoDB.** Mongodb.com, [s. l.], c2023. Disponível em: <https://www.mongodb.com/pricing>. Acesso em: jun. 2023.
NODE PACKAGE MANAGER - NPM. NPM: **npm documentation.** Docs.npmjs.com, [s. l.], Disponível em: <https://www.docs.npmjs.com/>. Acesso em: jan. 2023.
NODEJS. Node.js: **Documentation.** Nodejs.org, [s. l.], [2022?]. Disponível em: <https://nodejs.org/en/docs]>. Acesso em: jan. 2023.
NOMELINI, G. L.; GALANTE, G. **Reprodutibilidade de experimentos com uso de conteinerização. Anais do Congresso Latino-Americano de Software Livre e Tecnologias Abertas (Latinoware)**. Anais do XVIII Congresso Latino-Americano de Software Livre e Tecnologias Abertas. SBC, 13 out. 2021. Disponível em: <https://sol.sbc.org.br/index.php/latinoware/article/view/19919>. Acesso em: mai. 2023
JENSEN, N.; MIERS, C. Análise de desempenho dos orquestradores: **Kubernetes e Docker Swarm.** Anais da Escola Regional de Alto Desempenho da Região Sul (ERAD-RS). Anais da XXI Escola Regional de Alto Desempenho da Região Sul. SBC, 14 abr. 2021. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/14766>. Acesso em: mai. 2023.
ORACLE. **O Que É Um Banco De Dados Em Nuvem?**. Oracle.com, [s. l.], c2023. Disponível em: <https://www.oracle.com/br/database/what-is-a-cloud-database>. Acesso em: mai. 2023.
OCDE. PISA 2018 Results (Volume I): **What Students Know and Can Do.** Paris: OECD Publishing, 2019.
OCDE. **Recommendation on Principles and Good Practices for Financial Education and Awareness.** OECD, Paris, Jul/2005.
OLIVEIRA, R. et al. Gerenciamento de Dados em Nuvem: **Conceitos, Sistemas e Desafios.** In: Anais do XXXVIII Congresso da Sociedade Brasileira de Computação. Porto Alegre: SBC, 2018.
OLIVEIRA, R. et al. **Uma abordagem DevOps utilizando Kubernetes e Jenkins Pipeline para implantação automatizada em nuvem.** In: WORKSHOP DE COMPUTAÇÃO EM NUVENS E APLICAÇÕES (WCNA), 9., 2018, Campos do Jordão. Anais […]. Porto Alegre: Sociedade Brasileira de Computação, 2018.
OLIVEIRA, R. L. F.; PEDRON, C. D. Métodos Ágeis: **uma revisão sistemática sobre benefícios e limitações**. Brazilian Journal of Development, v. 7, n. 1, p. 4520–4534, 20 jan. 2021.

OSTI, E. W. A.; PEREIRA, R. B. DE O. **Análise de Desempenho do banco de dados SQL Server em Infraestruturas On Premise e Cloud.** Revista Eletrônica da Faculdade Invest de Ciências e Tecnologia, v. 5, n. 1, p. 14–14, 26 dez. 2021.
PAIXÃO, João Roberto da. **O que é SOLID: O guia completo para você entender os 5 princípios da POO**. 2019. Disponível em <https://medium.com/luizalabs/descomplicando-a-clean-architecture-cf4dfc4a1ac6>. Acesso em mai. 2023.
POSTGRESQL. **PostgreSQL Documentation.** Postgresql.org, [s. l.], c2023. Disponível em: <(https://www.postgresql.org/docs/)>. Acesso em: jan. 2023.
POSTGRESQL, 20.4. **Resource Consumption.** Postgresql.org, [s. l.], c2023. Disponível em: <https://www.postgresql.org/docs/15/runtime-config-resource.html>. Acesso em: jun. 2023.
PRISMA. **Top 11 Node.js ORMs, query builders & database libraries in 2022**. Prisma.io, [s. l.], 2022. Disponível em: <https://www.prisma.io/dataguide/database-tools/top-nodejs-orms-query-builders-and-database-libraries>
PROJECT MANAGEMENT INSTITUTE. **Um guia de conhecimento em gerenciamento de projetos.** Guia PMBOK. 6. ed. Pensilvânia, Project Management Institute, 2017.
QUILLICI, F.; SCHIAVON, L. **Uma Comparação entre sistemas MongoDB e PostgreSQL : Comparação entre sistemas MongoDB e PostgreSQL**. Congresso de Tecnologia - Fatec Mococa, v. 3, n. 2, 28 mai. 2021.
RED HAT INC - RED HAT. **O que é API REST?**. Redhat.com, [s. l.], 2020. Disponível em: <https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api>. Acesso em jul 2023
RIZARDI, B.; VICENTE, T. **Design ágil para inovação social e desenvolvimento,** 2020.
RUSSO, R. F. S. M.; SILVA, L. F. D.; LARIEIRA, C. L. C. **Do manifesto ágil à agilidade organizacional.** Revista de Gestão e Projetos, v. 12, n. 1, p. 1–10, 11 mar. 2021.
SANTOS, A. C.; SANTOS, A. L.; SILVA, F. S. C. **Desenvolvimento de uma API RESTful utilizando Spring Boot: um estudo de caso no contexto da Internet das Coisas.** In: Anais do XXXVIII Congresso da Sociedade Brasileira de Computação. Porto Alegre: SBC, 2018.
SCHWABER, K.; SUTHERLAND, J. **Guia do Scrum: Um guia definitivo para o Scrum: As regras do jogo,** 2017.
SANTOS, B.; ENDO, P. T.; SILVA, F. A. **Uma Avaliação de Desempenho de Containers Docker Executando Diferentes SGBDs Relacionais. Anais do Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance).** Anais XVIII Workshop em Desempenho de Sistemas Computacionais de Comunicação. SBC, 8 jul. 2019. Disponível em: <https://sol.sbc.org.br/index.php/wperformance/article/view/6467>. Acesso em: mai. 2023.
SERASA; Instituto Opinion Box. **Perfil e Comportamento do Endividamento Brasileiro.** 2022. Disponível em: <https://www.serasa.com.br/imprensa/pesquisa-de-endividamento-2022>. Acesso em: jun. 2023.
SILVA, A. C.; SANTOS, R. L.; ALMEIDA, R. A. **Uma análise comparativa entre as plataformas Docker Swarm e Kubernetes para orquestração de contêineres.** In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS

DISTRIBUÍDOS (SBRC), 38., 2020, Rio de Janeiro. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2020.

SOUZA, E. et al. **Desafios no uso da computação em nuvem para armazenamento seguro dos dados.** In: Anais do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais. Porto Alegre: SBC, 2017.

SOUZA, E. C.; OLIVEIRA, M. R. DE. COMPARATIVO ENTRE OS BANCOS DE DADOS MYSQL E MONGODB: **quando o MongoDB é indicado para o desenvolvimento de uma aplicação.** Revista Interface Tecnológica, v. 16, n. 2, p. 38–48, 21 dez. 2019.

STACKOVERFLOW. **Stack Overflow Developer Survey 2022**. 2022. Disponivel em <https://survey.stackoverflow.co/2022#top-paying-technologies-programming-scripting-and-markup-languages>. Acesso em: mai. 2023.

STOPA, G. R.; RACHID, C. L. SCRUM: **metodologia ágil como ferramenta de gerenciamento de projetos.** CES Revista, v. 33, n. 1, p. 302–323, 2 ago. 2019.

TANENBAUM, A.; STEEN, M. Distributed Systems: **Principles and Paradigms.** 2nd ed. Upper Saddle River: Prentice Hall PTR, 2007.

VALENTE, Marco Tulio. **Engenharia de Software Moderna.** 1ª edição. 2022

VARELLA, L. et al. **Orquestração de Aplicações de Computação de Alta Performance em Ambientes Cloud Conteinerizados**. Anais da Escola Regional de Alto Desempenho da Região Sul (ERAD-RS). SBC, 14 abr. 2021. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/14765>. Acesso em: mai. 2023.

WEB3TECHS. **Usage statistics of JavaScript for websites**. 2023. Disponível em <https://w3techs.com/technologies/details/pl-js>. Acesso em: jun. 2023.

ZAMMETTI, F. Modern Full-Stack Development: **Using TypeScript, React, Node.js, Webpack, and Docker**. 1st ed. Nova York: Apress, 1 jan. 2020.

ZARELLI, Guilherme Biff. **Descomplicando a Clean Architecture**. 2020. Disponível em <https://medium.com/luizalabs/descomplicando-a-clean-architecture-cf4dfc4a1ac6>. Acesso em: mar. 2023.