INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ

CAMPUS PICOS

CURSO ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ARMANDO PEREIRA DE SOUSA

GAMEMATE:

A Arquitetura de Backend por Trás de uma Biblioteca de Jogos Unificada

PICOS

2025

ARMANDO PEREIRA DE SOUSA

GAMEMATE:

A Arquitetura de Backend por Trás de uma Biblioteca de Jogos Unificada

Relatório técnico apresentado como exigência para aprovação na disciplina Trabalho de conclusão do Curso de Tecnologia em análise e desenvolvimento de sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Picos.

Orientador(a): Prof. MSc. Jader Anderson Oliveira de Abreu.

PICOS

2025

ARMANDO PEREIRA DE SOUSA

GAMEMATE:

A Arquitetura de Backend por Trás de uma Biblioteca de Jogos Unificada

Relatório técnico apresentado como exigência para aprovação na disciplina Trabalho de conclusão do Curso de Tecnologia em análise e desenvolvimento de sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Picos.

Aprovada em: ___/___/_____.

BANCA EXAMINADORA

_____

Prof. MSc. Jader Anderson Oliveira de Abreu (Orientador)

Instituto Federal do Piauí (IFPI)

_____

Prof. MSc. João Paulo Lima do Nascimento

Instituto Federal do Piauí (IFPI)

_____

Prof. MSc. Aislan Rafael Rodrigues de Sousa

Instituto Federal do Piauí (IFPI)

RESUMO

O mercado de jogos digitais enfrenta uma crescente fragmentação, com múltiplas plataformas de distribuição operando como ecossistemas fechados, o que obriga os jogadores a gerenciar diversas bibliotecas e autenticações dispersas. Este trabalho apresenta o desenvolvimento da arquitetura de backend do *GameMate*, uma solução de software projetada para unificar e gerenciar essas coleções em uma interface centralizada. A aplicação foi construída sobre uma arquitetura modular utilizando o *framework NestJS* e a linguagem *TypeScript*, integrando-se a serviços externos como *Steam*, *IGDB* e *SteamGridDB*. Para garantir escalabilidade e desempenho, adotou-se uma estratégia de processamento assíncrono baseada em filas (Bull) e cache em memória (Redis), suportada por um banco de dados relacional *PostgreSQL* e orquestrada via *Docker*. A validação do sistema incluiu testes automatizados ponta-a-ponta e testes de carga, que demonstraram latência interna mediana de 0,76ms e resiliência frente a limitações de *APIs* de terceiros. O projeto, que obteve o prêmio de melhor aplicativo na I Mostra de Informática (MINFO) do IFPI Campus Picos, confirmou a viabilidade técnica de agregar bibliotecas heterogêneas através de uma infraestrutura robusta e segura.

Palavras-chave: Agregador de Jogos. Arquitetura de Backend. NestJS. Processamento Assíncrono. Microsserviços.

ABSTRACT

The digital games market faces increasing fragmentation, with multiple distribution platforms operating as walled gardens, forcing players to manage scattered libraries and authentications. This work presents the development of the backend architecture for GameMate, a software solution designed to unify and manage these collections in a centralized interface. The application was built on a modular architecture using the NestJS framework and TypeScript language, integrating with external services such as Steam, IGDB, and SteamGridDB. To ensure scalability and performance, an asynchronous processing strategy based on queues (Bull) and in-memory caching (Redis) was adopted, supported by a PostgreSQL relational database and orchestrated via Docker. System validation included automated end-to-end tests and load tests, which demonstrated a median internal latency of 0.76ms and resilience against third-party API limitations. The project, which was awarded Best Application at the 1st Informatics Exhibition (MINFO) of IFPI Campus Picos, confirmed the technical feasibility of aggregating heterogeneous libraries through a robust and secure infrastructure.

Keywords: Game Library Aggregation. Backend Architecture. NestJS. Asynchronous Processing. Microservices.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface (Interface de Programação de Aplicações)
DER	Diagrama de Entidade-Relacionamento
GOG	Good Old Games
HTTP	Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto)
HTTPS	Hypertext Transfer Protocol Secure (Protocolo de Transferência de Hipertexto Seguro)
IGDB	Internet Game Database
JWT	JSON Web Token
ORM	Object-Relational Mapper (Mapeador Objeto-Relacional)
PaaS	Platform as a Service (Plataforma como Serviço)
PK	Primary Key (Chave Primária)
FK	Foreign key (Chave Secundária)
RF	Requisito Funcional
RNF	Requisito Não Funcional
UID	User Identifier (Identificador de Usuário)

SUMÁRIO

1 Introdução

O mercado de jogos digitais, em sua constante evolução, replica características da indústria cultural tradicional, como o alto custo de produção e o baixo custo de reprodução. Essa dinâmica impulsiona a busca por mercados globais e, consequentemente, leva a uma distribuição multiplataforma que, paradoxalmente, resulta em uma crescente fragmentação (Fragmentação no mercado de jogos digitais, 2014). Grandes empresas estabelecem seus próprios ecossistemas, ou "jardins murados" (*walled gardens*), para maximizar a monetização e reter usuários, muitas vezes através de acordos de exclusividade de conteúdo (Severin, 2018). Para o consumidor, o resultado é uma experiência digital "confusa, mal coordenada, que o força a aprender vários protocolos de pesquisa" (Planejamento, 2014]), onde é preciso gerenciar múltiplas aplicações, senhas e coleções de jogos dispersas.

A necessidade de centralizar essas informações é um problema recorrente, validado por trabalhos acadêmicos que identificam a "falta de ferramentas ou sites que centralizam informações de jogos e promoções de diferentes plataformas em um único local" como uma lacuna a ser preenchida (Game tracking - RIC-CPS, 2022). A solução para essa desorganização encontra um forte paralelo conceitual no planejamento de bibliotecas digitais, cujo objetivo sempre foi consolidar o acesso a coleções heterogêneas para que pudessem ser "descobertas e pesquisáveis através de uma única interface de usuário" (Planejamento de biblioteca digitais, 2014).

Embora já existam no mercado soluções de agregação, elas apresentam limitações. O *GOG Galaxy 2.0*, por exemplo, apesar de sua proposta inicial, sofre com a estagnação no desenvolvimento e a instabilidade de suas integrações comunitárias. O *Playnite*, por sua vez, é uma alternativa de código aberto robusta e flexível, mas sua dependência de que os lançadores originais permaneçam instalados e sua complexidade de configuração representam barreiras para uma experiência de usuário totalmente fluida.

A análise desse cenário revela uma clara oportunidade para o desenvolvimento de uma nova solução. A hipótese central deste trabalho é que é possível criar um agregador que supere as limitações dos concorrentes. O GameMate é um agregador que contará com uma arquitetura de *backend* superior, focada em segurança, interoperabilidade e escalabilidade. O *GameMate* não apenas

resolveria o problema prático da fragmentação, mas também se alinharia aos princípios de unificação e acesso simplificado já consolidados na teoria de bibliotecas digitais, oferecendo um valor tangível a milhões de jogadores.

A relevância prática e o potencial de inovação deste projeto foram, inclusive, validados preliminarmente no ambiente acadêmico. Durante a I Mostra de Informática (MINFO) do IFPI Campus Picos, o protótipo funcional do GameMate foi submetido à avaliação técnica e pública, sendo agraciado com o prêmio de 'Melhor Aplicativo'. Este reconhecimento reforça a hipótese de que a fragmentação das bibliotecas de jogos representa uma demanda real dos usuários e confirma a aderência da solução proposta às necessidades do mercado.

Este trabalho está estruturado para detalhar a concepção e o projeto técnico da solução. Primeiramente, são definidos os Objetivos. Em seguida, a seção de Tecnologias Envolvidas apresenta as ferramentas e *frameworks* selecionados para a implementação. O cerne do trabalho reside na seção de Modelagem do Projeto, que começa com o Levantamento de Requisitos para determinar as funcionalidades do sistema. Os Diagramas de Casos de Uso e de Classe, então, ilustram as interações e a estrutura do *software*. A Arquitetura do Sistema descreve a organização dos componentes de *backend*, e o Diagrama de Entidades-Relacionamentos especifica o modelo do banco de dados. Por fim, a seção de Interface exibe os protótipos visuais da aplicação.

1.1 Objetivos

Para nortear o desenvolvimento da pesquisa e a construção da solução proposta, foram definidos os seguintes objetivos

1.1.1 Geral

Desenvolver uma aplicação de *software*, denominada GameMate, com foco na construção de uma arquitetura de *backend* robusta e escalável, capaz de agregar e gerenciar bibliotecas de jogos de múltiplas plataformas de distribuição digital em uma interface unificada.

1.1.2 Específicos

- Analisar o cenário de fragmentação do mercado de jogos digitais e seu impacto na experiência do usuário para fundamentar a relevância da solução.
- Definir as soluções de agregação existentes (*GOG Galaxy 2.0*, *Playnite*, entre outras) para identificar suas arquiteturas, funcionalidades, pontos fortes, fraquezas e as lacunas de mercado.
- Definir os princípios de uma arquitetura de *backend* multicamadas, detalhando estratégias para integração de *APIs*, sincronização de dados e otimização de desempenho através de *cache*.
- Propor um modelo de dados unificado e um esquema de banco de dados (relacional ou objeto-relacional) capaz de harmonizar informações de fontes heterogêneas de forma consistente.
- Especificar um *framework* de segurança para o gerenciamento de dados sensíveis, com ênfase no armazenamento seguro e no ciclo de vida de *tokens* de autenticação de terceiros (*OAuth 2.0*).
- Implementar um protótipo funcional do *backend* da aplicação *GameMate* utilizando o *framework NestJS* e o *ORM Prisma*, desenvolvendo os módulos de autenticação de usuários, sincronização assíncrona de bibliotecas da *Steam* e as *APIs* para gerenciamento de jogos.

2.0 Tecnologia envolvidas

A arquitetura do projeto *GameMate* foi construída sobre um conjunto de tecnologias modernas e robustas, escolhidas para garantir escalabilidade, performance e uma boa experiência de desenvolvimento. As tecnologias são divididas nas seguintes categorias:
Backend:

- NestJS*: Framework* principal para o desenvolvimento do *backend*, utilizando a linguagem *TypeScript*. Foi escolhido pela sua arquitetura modular, sistema de injeção de dependência e por promover código organizado e testável.
- Prisma*: ORM (Object-Relational Mapper)* utilizado para a interação com o

banco de dados *PostgreSQL*. Ele garante a segurança de tipos *(type-safety)* entre o banco de dados e o código da aplicação.

- Bull / Bull Board: Sistema de filas de *jobs*, utilizado para executar tarefas de alto custo computacional e demoradas (como a sincronização de jogos) de forma assíncrona, sem bloquear a *API* principal. O *Bull Board* foi adicionado para o monitoramento visual das filas.
- Passport.js: *Middleware* de autenticação modular para *Node.js*. No projeto, foi especificamente utilizado com a estratégia *passport-steam* para implementar o fluxo de autenticação *OpenID* com a *Steam*.

Banco de Dados e Infraestrutura:

- PostgreSQL: Sistema de gerenciamento de banco de dados relacional, escolhido pela sua robustez, confiabilidade e suporte a tipos de dados avançados como arrays (*String*[]) utilizados pelo Prisma.
- Redis: Armazenamento de dados em memória de alta performance, utilizado para duas funções críticas:
    - Cache de API: Armazena temporariamente as respostas da *API* da *IGDB* para reduzir a latência e o uso da *API*.
    - Broker de Filas: Serve como *backend* para o sistema de *jobs* do *Bull*.
- Docker / Docker Compose: Plataforma de conteinerização utilizada para criar um ambiente de desenvolvimento local consistente e replicável, gerenciando os serviços de *PostgreSQL* e *Redis*.
- Render: Plataforma de nuvem (*PaaS*) utilizada para a hospedagem e *deploy* contínuo da aplicação *backend*.

APIs e Serviços Externos:

- Firebase Authentication: Serviço de autenticação do Google, utilizado para gerenciar o cadastro e login dos usuários, garantindo um processo seguro e desacoplado da aplicação principal.
- IGDB API: Fonte de dados principal para informações e metadados de jogos, como resumos, notas, géneros e plataformas.
- Steam Web API: Utilizada para o fluxo de autenticação *OpenID* e para buscar a lista de jogos e metadados básicos diretamente da biblioteca do usuário na *Steam*.
- SteamGridDB API: Fonte especializada e primária para a obtenção de arte de alta qualidade (capas/*grids*) para os jogos.

3.0 Modelagem do projeto

A modelagem do projeto constitui uma etapa fundamental na engenharia de software, servindo como o elo entre a definição das tecnologias, apresentada na seção anterior, e a implementação efetiva do sistema. Este capítulo detalha o planejamento arquitetural e funcional do GameMate, traduzindo a visão do produto em especificações técnicas precisas e documentadas.

O processo de modelagem aqui descrito visa assegurar que a solução atenda tanto às necessidades de interação do usuário quanto aos critérios de desempenho e escalabilidade do backend. Para isso, o capítulo estrutura-se inicialmente no levantamento formal de requisitos funcionais e não funcionais. Na sequência, são apresentados os artefatos visuais baseados na *Unified Modeling Language* (UML) e diagramas de arquitetura, abrangendo desde os casos de uso e estrutura de classes até a modelagem de dados e fluxos de sequência, fornecendo a base lógica para o desenvolvimento da aplicação.

3.1 Levantamento de requisitos

O levantamento de requisitos define formalmente o que o sistema deve fazer. Podemos dividi-lo em duas categorias principais: Requisitos Funcionais (RF), que descrevem as funcionalidades do sistema (o que ele faz), e Requisitos Não Funcionais (RNF), que descrevem as qualidades e restrições do sistema (como ele faz).

3.1.2 Requisitos funcionais

Os Requisitos Funcionais definem as funções que o sistema deve fornecer, descrevendo como ele deve reagir a entradas específicas e comportar-se em situações particulares. Segundo Sommerville (2019), estes requisitos variam desde declarações abstratas de alto nível até especificações detalhadas de funcionamento.

Para o projeto GameMate, a elicitação dos requisitos funcionais baseou-se na análise das lacunas de usabilidade identificadas nas ferramentas de agregação existentes, como o *GOG Galaxy* e o *Playnite*, conforme discutido na seção

introdutória. O (Quadro 1) apresenta a lista das funcionalidades essenciais definidas para o MVP (*Minimum Viable Product*) do sistema.

Quadro 1 – Requisitos Funcionais (RF)

ID	Nome do Requisito	Descrição
RF01	Cadastro de Usuário	O sistema deve permitir que um novo usuário se cadastre na plataforma utilizando uma conta do *Firebase Authentication*. O sistema deve criar um perfil correspondente no banco de dados local.
RF02	Autenticação de Usuário	O sistema deve permitir que um usuário existente se autentique na plataforma via *Firebase*, gerando um *token* de acesso (JWT).
RF03	Visualização de Perfil	Um usuário logado deve poder visualizar as informações do seu próprio perfil, incluindo nome, e-mail e estatísticas (ex: total de jogos, tempo de jogo).
RF04	Atualização de Perfil	Um usuário logado deve poder atualizar as informações editáveis do seu perfil (ex: nome de usuário).
RF05	Exclusão de Perfil	Um usuário logado deve poder excluir permanentemente a sua conta. A exclusão deve remover os dados do usuário tanto do *Firebase* quanto do banco de dados local.
RF06	Vinculação de Contas Externas	O sistema deve permitir que um usuário logado vincule sua conta de lojas de jogos externos (inicialmente a *Steam*) ao seu perfil *GameMate*.
RF07	Desvinculação de Contas Externas	O sistema deve permitir que um usuário logado remova a vinculação de uma conta de loja externa, o que também deve remover os jogos associados

ID	Nome do Requisito	Descrição
		àquela fonte da sua biblioteca.
RF08	Sincronização da Biblioteca de Jogos	Após vincular uma conta (ex: *Steam*), o sistema deve iniciar um processo em segundo plano para buscar a lista de jogos do usuário, enriquecer os dados com informações de APIs externas (*Steam*, *IGDB*, *SteamGridDB*) e salvá-los no banco de dados.
RF09	Re-sincronização da Biblioteca	Um usuário com uma conta vinculada deve poder disparar manualmente uma nova sincronização para atualizar sua biblioteca de jogos.
RF10	Visualização da Biblioteca de Jogos	Um usuário logado deve poder visualizar a sua biblioteca de jogos, com opções de filtro (por status, nome, loja) e paginação ("rolagem contínua").
RF11	Gestão de Status de Jogo (CRUD)	Um usuário logado deve poder atribuir e atualizar um status (ex: Jogando, Zerado, Platinado, etc.) para cada jogo na sua biblioteca.
RF12	Adição Manual de Jogo	Um usuário logado deve poder adicionar um jogo à sua biblioteca manualmente a partir da página de detalhes do jogo.
RF13	Remoção Manual de Jogo	Um usuário logado deve poder remover um jogo da sua biblioteca.
RF14	Busca Pública de Jogos	O sistema deve fornecer uma funcionalidade de busca para que qualquer usuário (logado ou não) possa pesquisar jogos no catálogo da plataforma (via *IGDB*).
RF15	Visualização de	Qualquer usuário deve poder visualizar uma página

ID	Nome do Requisito	Descrição
	Detalhes do Jogo	de detalhes de um jogo, com informações ricas como resumo, capa, géneros, plataformas, etc.

Fonte: Elaborado pelo autor (2025).

Em resumo, os requisitos funcionais direcionam o *backend* a focar em quatro pilares essenciais:

- Gerenciamento de Usuários e Contas, abrangendo quase todo o ciclo de vida do perfil, desde o cadastro (RF01) até os itens (RF02, RF04 e RF05)
- Integração com Serviços Externos, com destaque para o fluxo de vinculação com a *Steam* (RF06) e a re-sincronização de dados (RF09)
- Processamento Assíncrono de Dados, através de um robusto sistema de filas para orquestrar a sincronização e o enriquecimento da biblioteca de jogos (RF08)
- E a Gestão da Biblioteca Pessoal, permitindo ao usuário visualizar, filtrar e gerenciar os status de cada um dos seus jogos (RF10, RF11).

3.1.3 Requisitos não funcionais

Enquanto os requisitos funcionais detalham o que o sistema faz, os Requisitos Não Funcionais (RNF) estabelecem restrições aos serviços ou funções oferecidos pelo sistema, incluindo restrições de tempo, processo de desenvolvimento e padrões. De acordo com Pressman e Maxim (2016), esses requisitos são frequentemente críticos, pois o não atendimento a um deles (como confiabilidade ou segurança) pode inutilizar todo o sistema, independentemente de suas funcionalidades.

Para garantir a robustez da arquitetura proposta, foram definidos critérios rigorosos de desempenho, segurança e manutenibilidade. O (Quadro 2) detalha os Requisitos Não Funcionais que nortearam as decisões arquiteturais do projeto.

Quadro 2 – Requisitos Não Funcionais (RNF)

ID	Nome do requisito	Descrição
RNF01	Performance	A *API* deve responder a requisições de leitura (ex: *GET* /games/:id) em menos de 500ms para dados já cacheados ou no banco. Processos demorados, como a sincronização, devem ser assíncronos e não devem bloquear a *API* principal.
RNF02	Escalabilidade	A arquitetura deve suportar um número crescente de usuários e jogos sem degradação de performance, utilizando um sistema de filas (*Bull/Redis*) e um banco de dados relacional (*PostgreSQL*).
RNF03	Segurança	A autenticação deve ser gerenciada pelo *Firebase*. Todas as rotas que manipulam dados de usuários devem ser protegidas e validar um token *JWT*. As senhas dos usuários nunca devem ser armazenadas no sistema.
RNF04	Usabilidade	A interface (*Frontend*) deve ser intuitiva. A vinculação de contas e a sincronização devem fornecer *feedback* claro ao usuário sobre o estado do processo.
RNF05	Portabilidade	A aplicação *backend* deve ser conteinerizada com *Docker*, garantindo que possa ser executada de forma consistente em qualquer ambiente (desenvolvimento, produção).
RNF06	Manutenibilidade	O código-fonte deve ser modular e seguir os padrões de arquitetura do *NestJS*, com uma clara separação entre controladores,

ID	Nome do requisito	Descrição
		serviços e repositórios para facilitar a manutenção e a adição de novas funcionalidades.

Fonte: Elaborado pelo autor (2025).

Em suma, os requisitos não funcionais estabelecem os pilares de qualidade da arquitetura do *backend*. O foco principal está na Performance e Escalabilidade, garantidas pelo uso de processamento assíncrono com filas (RNF01, RNF02) para tarefas demoradas. A Segurança é um pilar crítico, delegando a autenticação a um serviço especializado como o *Firebase* e protegendo rigorosamente todos os endpoints de dados do usuário (RNF03). Por fim, a Manutenibilidade e a Portabilidade são asseguradas pela adoção da arquitetura modular do *NestJS* e pela conteinerização completa do ambiente com *Docker* (RNF05, RNF06), garantindo que o sistema seja robusto, fácil de manter e de implantar em qualquer ambiente.

3.2 Diagramas de casos de uso

Os Diagramas de Casos de Uso são utilizados para visualizar as diferentes maneiras como um ator (um usuário ou outro sistema) interage com as funcionalidades do *software*. Eles fornecem uma visão de alto nível do que o sistema faz e quem o utiliza. Para o GameMate, dividimos os casos de uso em três diagramas principais para maior clareza: Gerenciamento de Conta (Figura 1), Gerenciamento da Biblioteca (Figura 2) e Descoberta de Jogos (Figura 3).

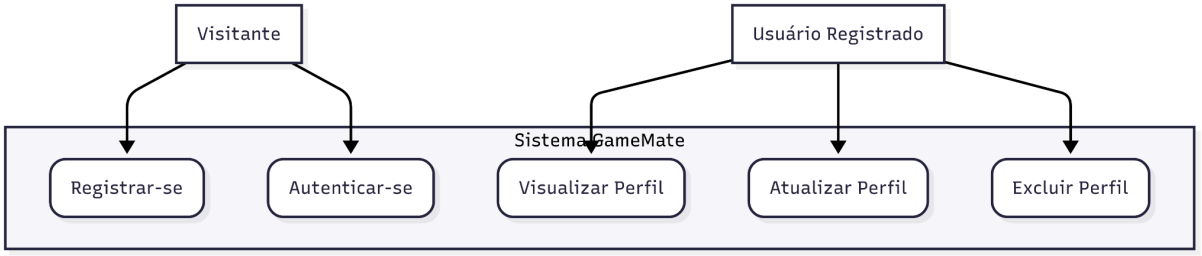
3.2.1 Gerenciamento de conta e perfil

O diagrama (Figura 1) delimita as fronteiras de acesso inicial ao sistema, segregando as permissões entre dois atores distintos. O ator **'**Visitante**'** representa o usuário não autenticado, cujas interações restringem-se aos pontos de entrada da aplicação: o registro de novos dados e a autenticação.

Em contrapartida, o 'Usuário Registrado' representa o ator que transpôs a barreira de segurança; ele detém permissões exclusivas para invocar os casos de

uso de manutenção de conta, permitindo a visualização, atualização e a exclusão definitiva dos seus dados de perfil no sistema.

Figura 1 – Gerenciamento de conta e perfil

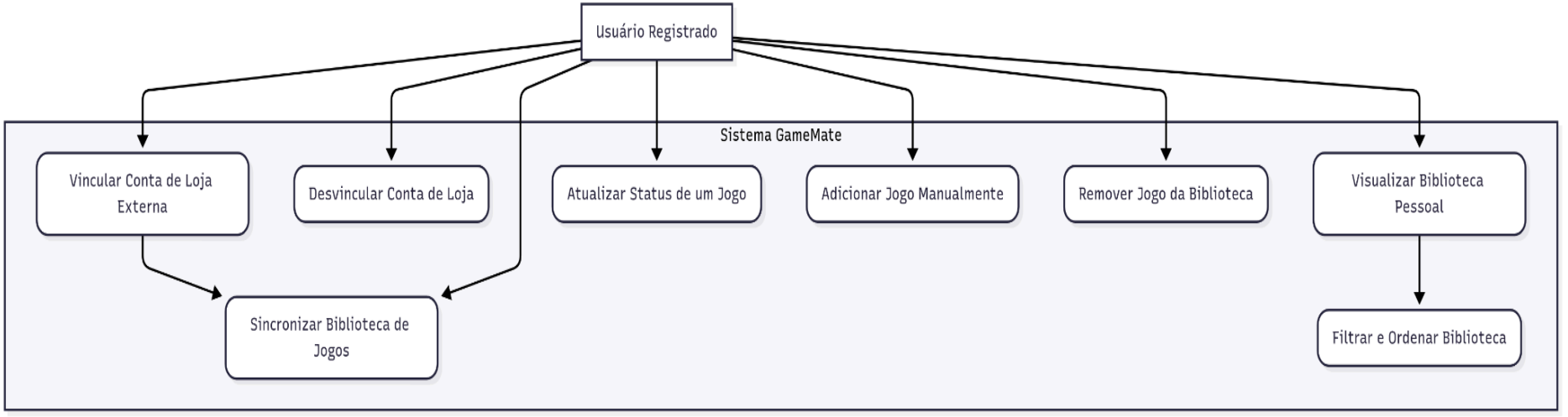


Fonte: Elaborado pelo autor (2025).

3.2.2 Gerenciamento da biblioteca e contas externas

Este diagrama (Figura 2) foca nas funcionalidades principais disponíveis para um **Usuário Registrado**. Ele detalha como o usuário interage com sua biblioteca pessoal de jogos, gerencia o *status* de cada jogo e administra a vinculação com contas de lojas externas, como a *Steam*. A sincronização de jogos é mostrada como uma consequência da vinculação da conta.

Figura 2 – Gerenciamento da biblioteca de jogos

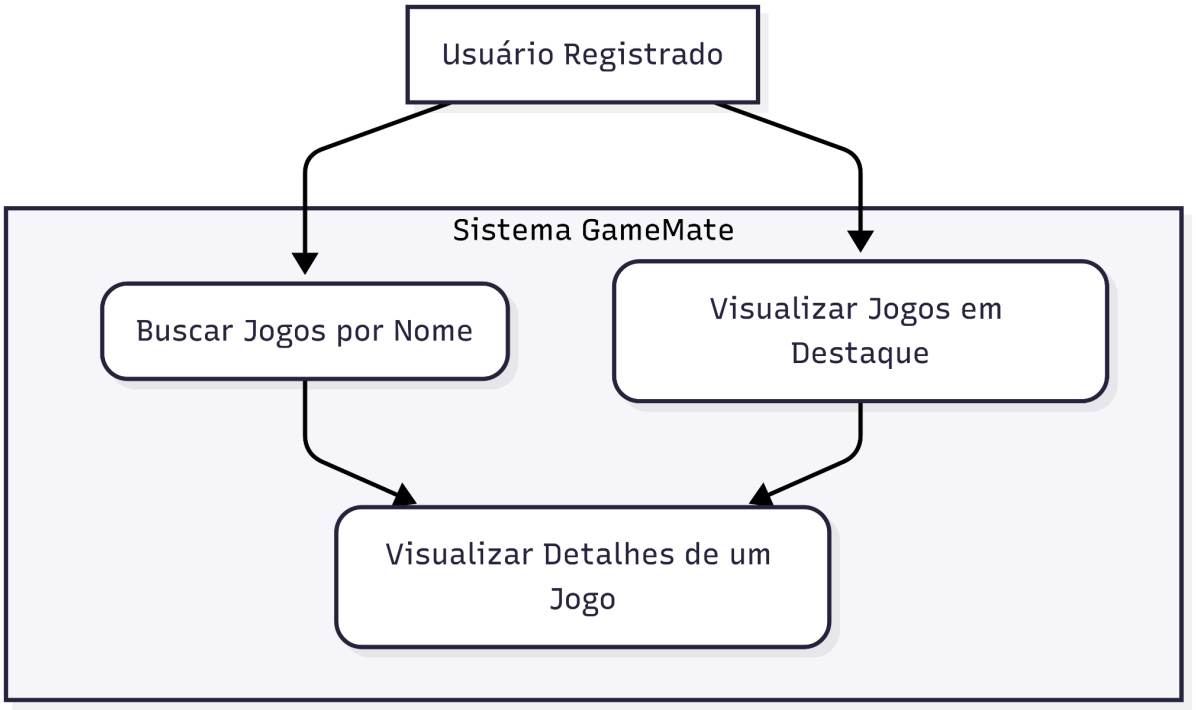


Fonte: Elaborado pelo autor (2025)

3.2.3 Descoberta de jogos

O diagrama de casos de uso apresentado na Figura 3 mapeia as funcionalidades de descoberta de conteúdo, essenciais para o engajamento na plataforma. Ele ilustra as vias pelas quais o 'Usuário Registrado' interage com o catálogo global do sistema (integrado à API da IGDB). O fluxo permite tanto a busca ativa, através do caso de uso 'Buscar Jogos por Nome', quanto a navegação passiva via 'Visualizar Jogos em Destaque'. Ambas as interações convergem para o caso de uso 'Visualizar Detalhes de um Jogo', que exibe os metadados completos e habilita as operações de gerenciamento de biblioteca.

Figura 3 – Gerenciamento da biblioteca de jogos



Fonte: Elaborado pelo autor (2025)

3.3 Diagramas de classes

O Diagrama de Classes é uma representação estática fundamental na modelagem orientada a objetos. Segundo Sommerville (2019), este diagrama descreve os tipos de objetos do sistema e os relacionamentos estáticos que existem entre eles, detalhando atributos e operações, independentemente de como esses objetos interagem em tempo de execução.

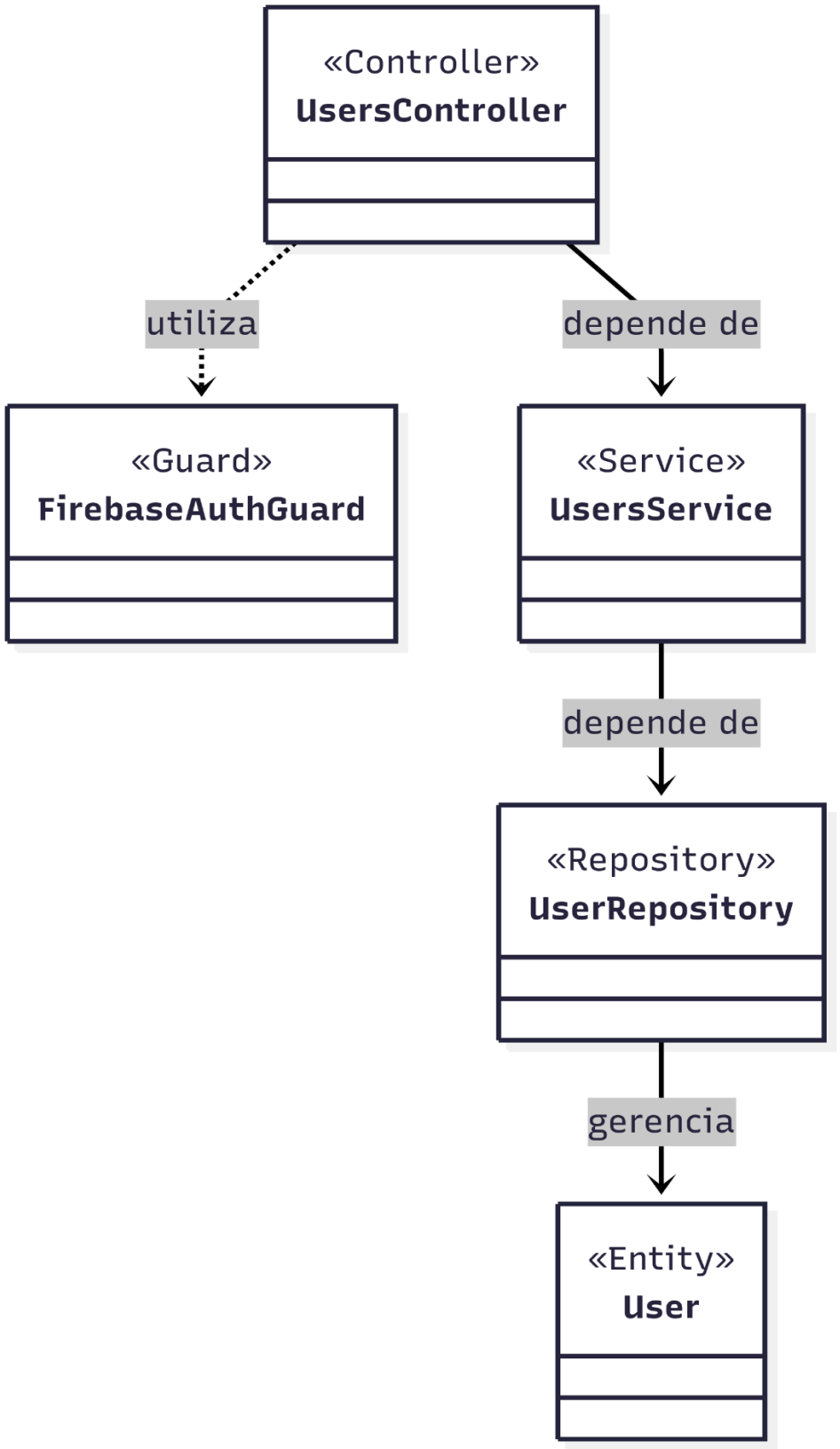
Dada a complexidade da arquitetura do GameMate, optou-se por segmentar a representação em visualizações focadas, evitando a sobrecarga de informações em um único modelo. A seguir, são apresentados três diagramas que ilustram as principais facetas da arquitetura do *backend*, permitindo uma análise granular dos módulos do sistema.

3.3.1 Arquitetura de um módulo (Exemplo: UsersModule)

A estrutura vertical apresentada na Figura 4 evidencia o princípio da Separação de Preocupações (Separation of Concerns), onde o fluxo de dependência ocorre de forma hierárquica e unidirecional. No topo, o UsersController gerencia exclusivamente a camada de apresentação (HTTP) e a segurança (via FirebaseAuthGuard), delegando a execução das regras de negócio para o UsersService. Este, por sua vez, centraliza a lógica da aplicação de forma agnóstica ao protocolo, sendo instanciado automaticamente através do sistema de Injeção de Dependência do NestJS.

Na camada inferior, a persistência é abstraída pelo UserRepository, que atua como mediador entre a lógica de negócio e o banco de dados. Ele utiliza o Prisma ORM para manipular a entidade User, garantindo que as operações de dados sejam tipadas e seguras. Essa organização em camadas desacopla a infraestrutura de dados da interface da API, facilitando a manutenção, a escalabilidade e a realização de testes isolados em cada componente.

Figura 4 – Arquitetura Principal de um Módulo (Exemplo: UsersModule)

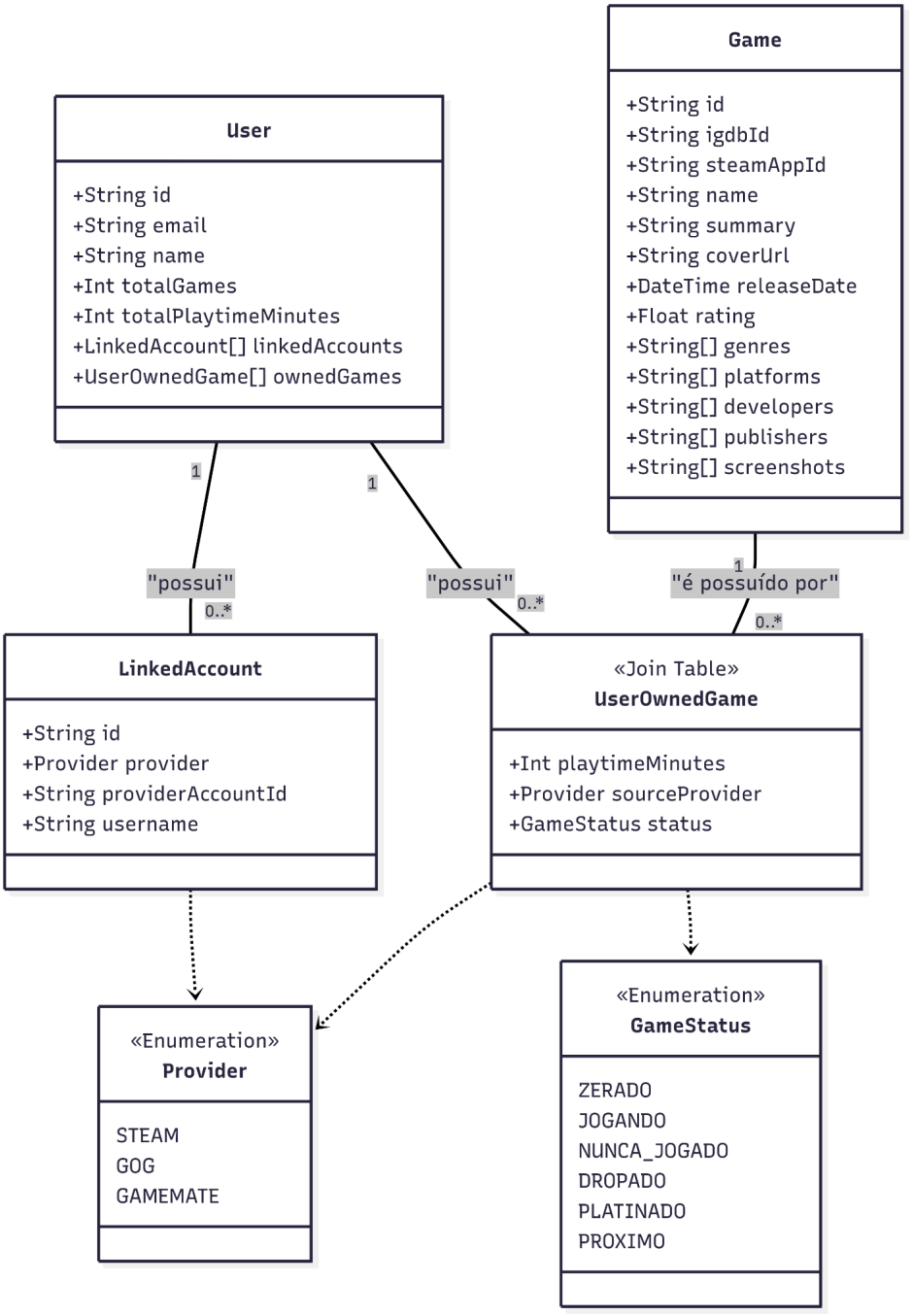


Fonte: Elaborado pelo autor (2025)

3.3.2 Modelo de entidades (banco de dados)

Este diagrama (Figura 5) foca exclusivamente nos modelos de dados e nas suas relações, servindo como uma representação visual direta do *schema.prisma*. Ele clarifica como as tabelas *User*, *Game*, *LinkedAccount* e a tabela de junção *UserOwnedGame* se conectam, além de mostrar o uso dos *Enums* para padronização.

Figura 5 – Modelo de Entidades (Banco de Dados)



Fonte: Elaborado pelo autor (2025)

3.3.3 Arquitetura de sincronização e enriquecimento

Este diagrama (Figura 6) ilustra a arquitetura assíncrona do sistema, que é uma das suas partes mais complexas e importantes. Ele mostra como os *Workers* (processadores de fila) interagem com os *Services* para buscar dados de múltiplas *APIs* externas (*Steam*, *IGDB*, *SteamGridDB*) e persistir os dados enriquecidos no banco através dos *Repositories*, tudo em segundo plano.

Figura 6 – Arquitetura de Sincronização e Enriquecimento

Fonte: Elaborado pelo autor (2025)

3.4 Arquitetura do sistema

A arquitetura do sistema GameMate foi projetada para ser modular, escalável

e resiliente, utilizando uma abordagem moderna de desenvolvimento de *backend*. Embora a aplicação seja implementada como um monólito modular, ela adota princípios de arquitetura de microsserviços, como a separação de responsabilidades e a comunicação assíncrona para tarefas pesadas.

3.4.1 Visão geral da arquitetura

A arquitetura é dividida em três camadas principais, seguindo o padrão *Layered Architecture*:

- Camada de Apresentação (API): Representada pelos *Controllers*. Esta camada atua como o ponto de entrada das requisições na aplicação. É responsável por receber as requisições *HTTP*, validar os dados de entrada através de *DTOs* e orquestrar a resposta. A segurança é garantida por *Guards*, como o *FirebaseAuthGuard*.
- Camada de Negócios (Lógica): Representada pelos *Services*. É aqui que reside a lógica de negócio principal da aplicação. Os serviços orquestram as operações, manipulam os dados e se comunicam com outras partes do sistema, como os repositórios e serviços externos.
- Camada de Acesso a Dados (Persistência): Representada pelos *Repositories*. Esta camada abstrai a comunicação com o banco de dados. Ela utiliza o *Prisma ORM* para executar queries no banco de dados *PostgreSQL*, garantindo que a lógica de negócio não precise de conhecer os detalhes da implementação do banco.

3.4.2 Processamento Assíncrono com Filas

Para lidar com tarefas demoradas e pesadas, como a sincronização da biblioteca de jogos da Steam, a arquitetura adota um padrão Produtor-Consumidor utilizando filas de *jobs*.

- Produtor: A API principal (ex: AuthController) atua como produtora, adicionando "*jobs*" (tarefas) a uma fila no *Redis*.
- Consumidor (Worker): Processos de background dedicados (*SteamSyncProcessor* e *GameEnrichmentProcessor*) atuam como consumidores. Eles "monitoram" a fila, consomem os (*jobs)* e os executam de

forma assíncrona, sem impactar a performance da *API* principal.

- Tecnologia: A biblioteca *Bull* é usada para gerenciar as filas sobre a infraestrutura do *Redis*.

3.4.3 Integração com Serviços Externos

O sistema foi projetado para se integrar com várias *APIs* externas, cada uma com um serviço dedicado para encapsular a sua lógica de comunicação:

- Firebase: Para autenticação de usuários (verificação de *tokens*).
- Steam: Para vinculação de contas (OpenID) e busca de dados de jogos.
- IGDB: Para busca e enriquecimento de metadados de jogos.
- SteamGridDB: Para enriquecimento de arte de alta qualidade (capas).

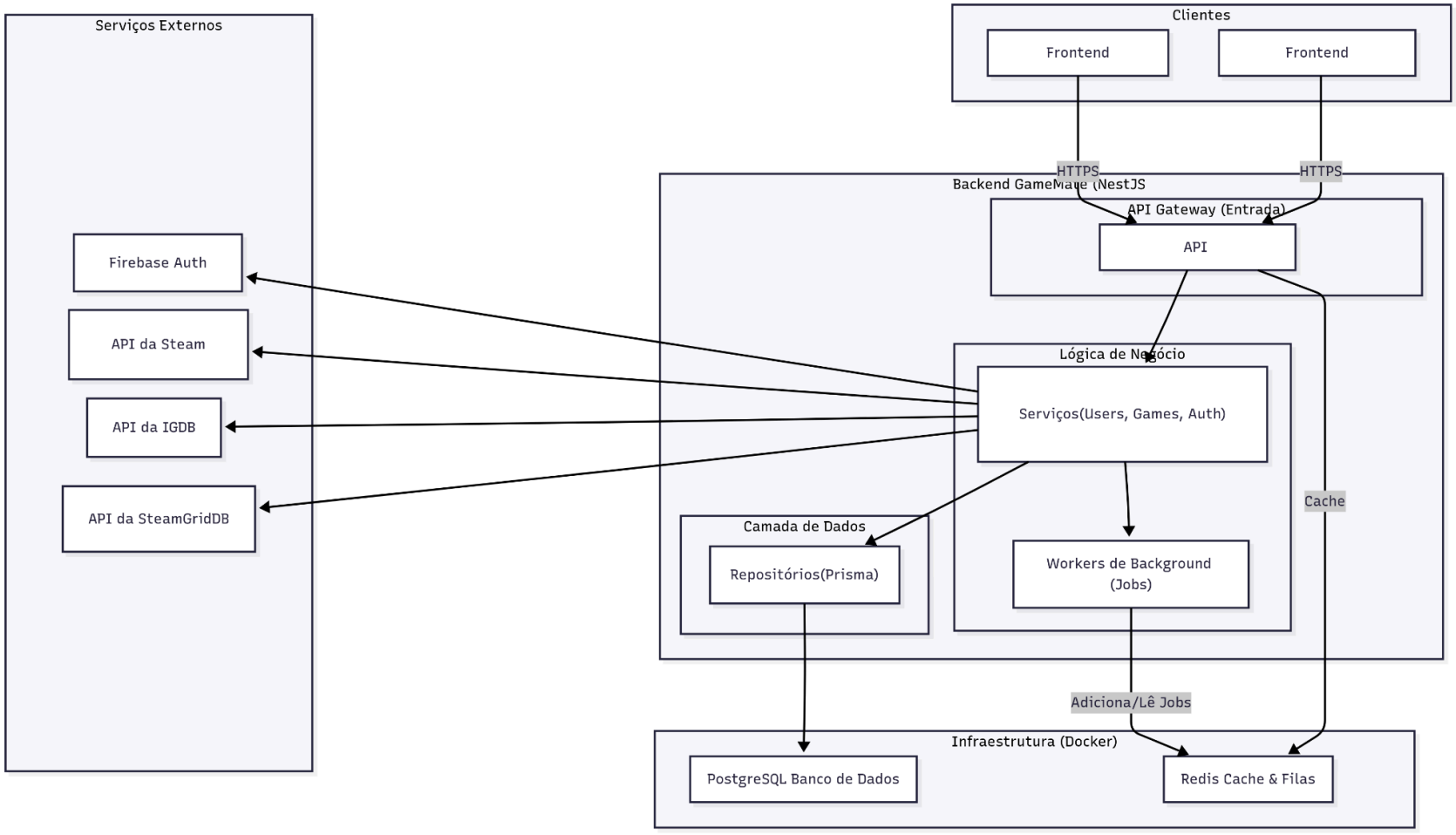
3.4.4 Diagrama de Arquitetura do Sistema

O diagrama (Figura 7) ilustra a visão macroscópica da solução, evidenciando o fluxo de comunicação desde a solicitação do cliente até a persistência dos dados. Na camada superior, observam-se os clientes (Frontend) que se comunicam com o backend via protocolo HTTPS seguro. As requisições são recebidas pelo *API Gateway*, que atua como ponto de entrada e roteamento para a camada interna da aplicação *NestJS*.

No núcleo do sistema, a 'Lógica de Negócio' centraliza as regras da aplicação através dos serviços (*Services*), orquestrando a interação com dois vetores principais: os serviços externos e a infraestrutura de dados. À esquerda, nota-se a integração com APIs de terceiros (*Firebase, Steam, IGDB e SteamGridDB*), fundamentais para a autenticação e o enriquecimento de metadados.

A parte inferior do diagrama destaca a estratégia de processamento e persistência. Enquanto os dados estruturados são gerenciados pelos repositórios e armazenados no banco de dados relacional *PostgreSQL* , as operações custosas são desviadas para os *Workers de Background*. O Redis desempenha aqui um papel duplo e crítico: atua como *broker* de mensagens para a fila de *jobs* assíncronos e como camada de *cache* para otimizar o tempo de resposta das consultas frequentes.

Figura 7 – Diagrama Geral da Arquitetura do Sistema



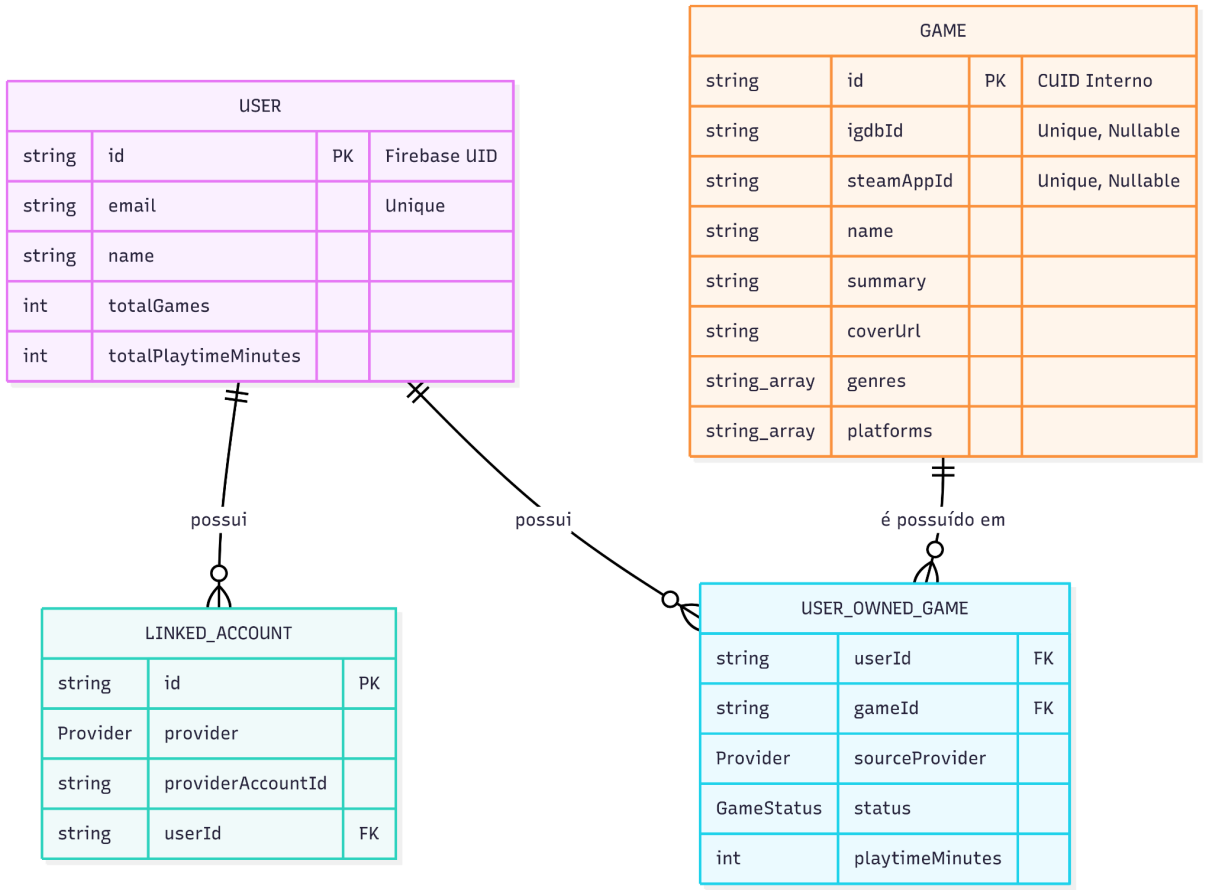
Fonte: Elaborado pelo autor (2025)

3.5 Diagrama de Entidade-Relacionamento

A modelagem de dados é um passo crítico no design de software, pois define a estrutura estática da informação que o sistema deve gerenciar. Segundo Pressman e Maxim (2016), o Diagrama de Entidade-Relacionamento (DER) permite representar graficamente os objetos de dados, descrever os atributos que definem suas propriedades e estabelecer os relacionamentos que conectam esses objetos, servindo como base para a construção física do banco de dados.

Para o sistema GameMate, o diagrama apresentado na Figura 8 reflete a estrutura definida no arquivo schema.prisma e implementada no banco de dados PostgreSQL. O modelo ilustra as interações entre as quatro entidades centrais: *USER*, *GAME*, *LINKED_ACCOUNT* e a tabela associativa *USER_OWNED_GAME*, detalhando como a persistência dos dados assegura a integridade das bibliotecas dos usuários.

Figura 8 – Diagrama de Entidade-Relacionamento

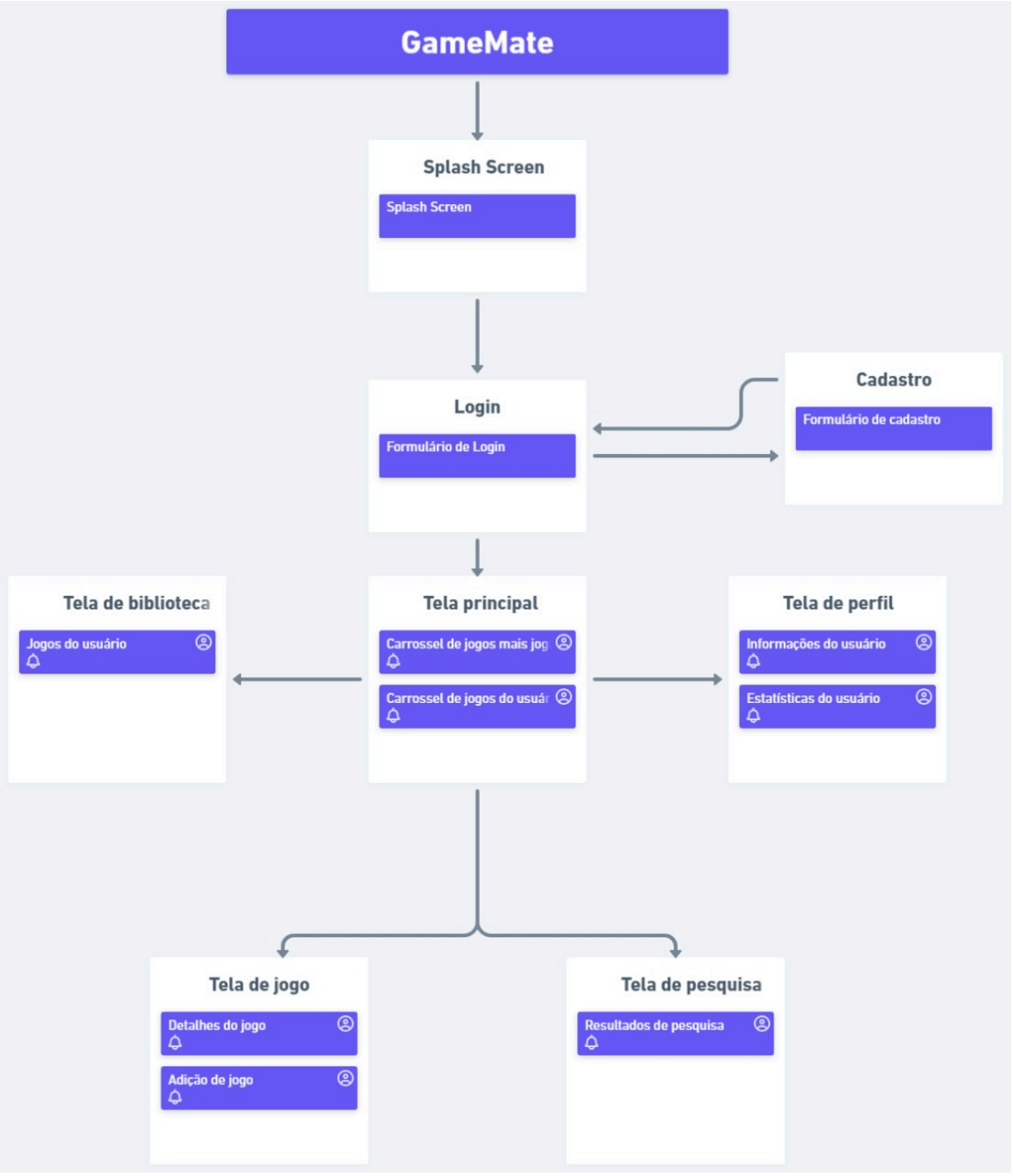


Fonte: Elaborado pelo autor (2025).

3.6 Interface

A interface da aplicação GameMate foi projetada para oferecer um fluxo de navegação claro e intuitivo, garantindo que o usuário possa aceder a todas as funcionalidades de forma eficiente. O fluxograma a seguir (Figura 9) ilustra a arquitetura da informação e a jornada do usuário através das principais telas do sistema.

Figura 9 – Fluxograma de Navegação da Interface



Fonte: Elaborado pelo autor (2025).

O fluxo de interação do usuário inicia-se com uma *Splash Screen*, que direciona para a tela de *Login*. Nela, o usuário pode autenticar-se ou navegar para a tela de Cadastro para criar uma nova conta.

Uma vez autenticado, o usuário acede à Tela Principal, que funciona como um dashboard central. Esta tela apresenta carrosséis de jogos, como os mais jogados e os da própria biblioteca do usuário, e serve como ponto de partida para as três áreas principais da aplicação:

- Tela de Perfil: Onde o usuário pode visualizar e gerir as suas informações pessoais e estatísticas de jogo.
- Tela de Jogo: A página de detalhes de um jogo específico, onde é possível ver todas as suas informações e adicioná-lo à biblioteca.
- Tela de Pesquisa: Uma funcionalidade para buscar e descobrir novos jogos no catálogo da plataforma.
- Tela de biblioteca: Uma página destinada a listar todos os jogos que o usuário possui, com funcionalidades como filtragem por lojas, *status* e busca por nome do jogo.

3.7 Diagrama de Sequência

Os diagramas a seguir exemplificam um dos fluxos principais do sistema. A veiculação da conta da Steam do usuário ao GameMate.

3.7.1 Vinculação de Conta (Interação do Usuário)

Este diagrama (Figura 10) mostra o fluxo de interação síncrona que ocorre quando um usuário decide vincular sua conta da *Steam* à sua conta GameMate. O processo foi desenhado para ser seguro e transparente, utilizando o padrão *OpenID* para delegação de autenticação e sessões de servidor para manter o estado entre os redirecionamentos.

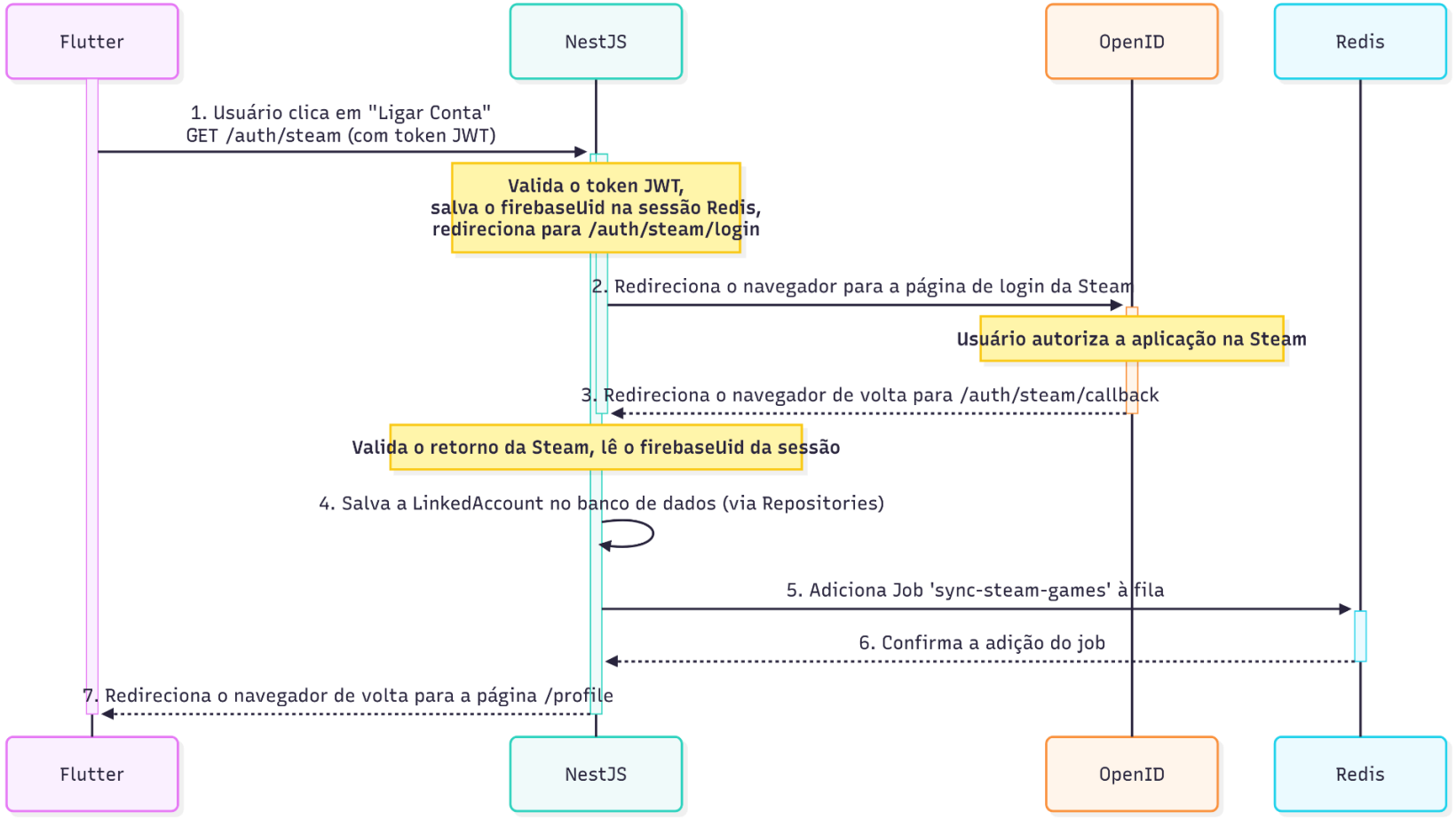
O fluxo detalhado, passo a passo, é o seguinte:

1. Início da Ação (Frontend): O usuário, já autenticado na aplicação GameMate, clica no botão "Vincular Conta Steam". O frontend realiza uma requisição para o backend (GET /auth/steam), enviando o *token* de autenticação (JWT) do

usuário para identificá-lo de forma segura.

2. Preparação e Redirecionamento (Backend): A *API NestJS* recebe a requisição, valida o *token JWT* para confirmar a identidade do usuário e armazena temporariamente o ID deste usuário (*firebaseUid*) em uma sessão no *Redis*. Esta etapa é crucial para garantir a persistência do estado da sessão do usuário durante o processo. Em seguida, o *backend* redireciona o navegador do usuário para a página de login oficial da *Steam*.
3. Autorização na Steam: O usuário interage diretamente com a plataforma da *Steam*, onde insere suas credenciais e autoriza a aplicação GameMate a aceder às suas informações básicas de perfil e biblioteca de jogos.
4. Callback e Validação (Backend): Após a autorização, a Steam redireciona o navegador do usuário de volta para o endpoint de callback da API (/auth/steam/callback). O backend recebe esta requisição, valida os dados retornados pela Steam e recupera o ID do usuário que estava guardado na sessão do Redis, garantindo que a conta Steam seja vinculada ao usuário correto.
5. Persistência e Disparo do Job (Backend): Com a confirmação da identidade, o backend salva permanentemente a nova vinculação na tabela *LinkedAccount* do banco de dados *PostgreSQL*. Imediatamente após, ele adiciona um novo "*job*" (sync-steam-games) na fila de processamento do *Redis*, instruindo os *workers* de segundo plano a iniciarem a sincronização da biblioteca de jogos daquele usuário.
6. Finalização do Fluxo: Por fim, o *backend* envia uma resposta final ao navegador, redirecionando o usuário de volta para a sua página de perfil na aplicação, onde ele poderá ver a confirmação da vinculação. Todo o processo de busca e enriquecimento dos jogos ocorrerá a partir deste momento em segundo plano, sem impactar a experiência de navegação do usuário.

Figura 10 – Diagrama de Sequência 1: Vinculação de Conta



Fonte: Elaborado pelo autor (2025)

3.7.2 Sincronização e Enriquecimento (Processo em Segundo Plano)

Este diagrama (Figura 11) detalha o processo assíncrono que ocorre em segundo plano, iniciado imediatamente após a vinculação da conta do usuário ser confirmada. O seu objetivo é popular a biblioteca do usuário sem que ele precise esperar, garantindo uma experiência de usuário fluida.

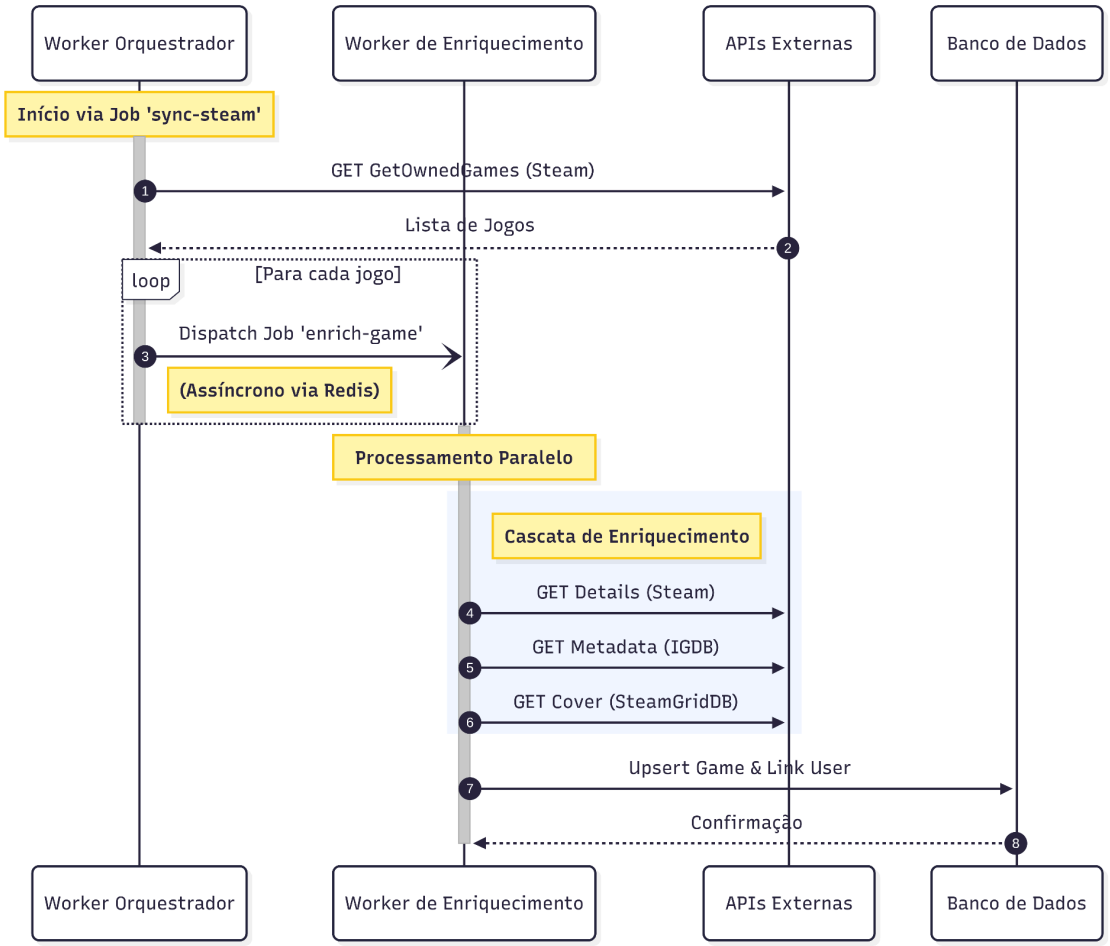
Para alcançar isso, o sistema implementa um padrão de design conhecido como *Orquestrador-Worker*, que divide uma tarefa grande e complexa em muitas tarefas pequenas e gerenciáveis que podem ser executadas em paralelo. O fluxo detalhado, passo a passo, é o seguinte:

1. Início do Processo (Fila de Orquestração): O processo começa quando um job do tipo *sync-steam-games* é colocado na fila de orquestração. Este evento atua como o gatilho inicial disparado pela *API* no final do fluxo de vinculação da conta.
2. Ação do Orquestrador: Um *worker* especializado, o *SteamSyncProcessor* (o Orquestrador), consome este *job*. A sua única responsabilidade é realizar uma única chamada à *API* da *Steam* para obter a lista completa de todos os jogos que o usuário possui.
3. Distribuição de Tarefas: Em vez de processar todos os jogos de uma vez, o Orquestrador atua como um controlador de fluxo: para cada jogo na lista recebida da *Steam*, ele cria e dispara um novo *job*, muito menor e mais específico (*enrich-single-game*), para uma segunda fila, a Fila de Enriquecimento. Se um usuário tiver 500 jogos, o Orquestrador criará 500 *jobs* individuais.
4. Processamento Paralelo (Workers de Enriquecimento): Vários *GameEnrichmentProcessors* (os *Workers*) monitoram a Fila de Enriquecimento. Cada *worker* pega um único job (*enrich-single-game*) e executa a "Fluxo de enriquecimento de Dados" para aquele jogo específico:
    a. Passo 5-7: O worker busca e consolida metadados de múltiplas fontes: primeiro a *API* da *Steam* para detalhes, depois a *API* da *IGDB* para informações complementares (como notas e géneros), e finalmente a *API* da *SteamGridDB* para a arte da capa de alta qualidade.
    b. Passo 8-11: Com todos os dados em mãos, o *worker* interage com o banco de dados *PostgreSQL* para primeiro criar ou atualizar o jogo no

catálogo global (*Game*) e, em seguida, criar o registo que vincula aquele jogo àquele usuário específico (*UserOwnedGame*), armazenando seu tempo de jogo e status.

A principal vantagem desta arquitetura é a sua resiliência e escalabilidade. O processamento de centenas de jogos é feito em paralelo pelos *workers*, tornando o tempo total muito menor. Além disso, se o enriquecimento de um jogo falhar por qualquer motivo, apenas aquele *job* específico falha, sem interromper o processamento dos demais itens da biblioteca do usuário.

Figura 11 – Diagrama de Sequência 2: Sincronização e Enriquecimento



Fonte: Elaborado pelo autor (2025).

4.0 Software

O software desenvolvido, denominado GameMate, cujo link para acesso é (https://gamemate-backend-dev.onrender.com) e o link para seu repositório é (https://github.com/CondeArmand/gamemate-backend.git), foi concretizado como uma *API RESTful* robusta, construída sobre a plataforma *Node.js* e estruturada com o *framework NestJS*. A arquitetura final do sistema reflete fielmente os requisitos não funcionais de modularidade e escalabilidade definidos nas etapas iniciais do projeto.

A implementação seguiu rigorosamente o padrão de arquitetura modular. Cada domínio da aplicação (Usuários, Jogos, Autenticação) foi encapsulado em módulos independentes, garantindo baixo acoplamento e alta coesão. A estrutura de diretórios consolidada no projeto final foi:

- src/app.module.ts: Módulo raiz responsável pela orquestração dos demais componentes.
- src/modules/: Diretório onde foram implementados os sub-módulos de domínio, contendo os *Controllers* (apresentação), *Services* (regras de negócio) e *Repositories* (acesso a dados).
- src/common/: Camada transversal contendo utilitários, decoradores e filtros de exceção padronizados.
- prisma/: Diretório contendo o esquema definitivo do banco de dados e o histórico de migrações aplicadas.

Todo o desenvolvimento foi realizado em linguagem *TypeScript*, o que assegurou a integridade dos dados através de tipagem estática. O gerenciamento de pacotes e dependências foi executado através do NPM (*Node Package Manager*).

4.1 Implantação

A implantação (*deploy*) do GameMate foi executada com sucesso, garantindo a disponibilidade do sistema em um ambiente de produção estável e acessível. A estratégia adotada baseou-se inteiramente na conteinerização para assegurar a consistência entre os ambientes de desenvolvimento e produção.

Para o encapsulamento da aplicação, foi configurado um Dockerfile otimizado utilizando uma imagem base *Node.js Alpine* (versão leve). Este arquivo foi

responsável por automatizar a instalação das dependências, a geração do cliente *Prisma* e a transpilação do código *TypeScript* para *JavaScript*.

A orquestração dos serviços foi realizada através do *Docker Compose*, que instanciou e gerenciou a comunicação entre os três contêineres fundamentais da infraestrutura:

- API GameMate: O serviço *backend* principal.
- PostgreSQL: O banco de dados relacional onde os dados persistentes foram armazenados.
- Redis: O banco de dados em memória utilizado para o gerenciamento das filas de processamento (*Bull*) e *cache* de performance.

O *deploy* final foi efetivado na plataforma de nuvem *Render* (*PaaS*). Foi configurado um fluxo de Entrega Contínua (*Continuous Deployment*), onde a aplicação foi automaticamente construída e atualizada a partir do repositório de código. Todas as variáveis de ambiente sensíveis (chaves de *API*, credenciais de banco) foram configuradas de maneira segura no painel de administração da plataforma, garantindo a segurança da implantação.

4.2 Testes

A garantia da qualidade de software é uma atividade essencial que deve ser aplicada em todas as fases do processo de engenharia, visando descobrir erros antes que o produto chegue ao usuário final. Segundo Pressman e Maxim (2016), os testes de software são elementos críticos para a garantia da qualidade e representam uma revisão final das especificações, do projeto e da codificação. Com a finalização do desenvolvimento do GameMate, foi executada uma estratégia de testes abrangente.

A abordagem adotada baseou-se no conceito da "Pirâmide de Testes" defendida por Fowler (2012), que sugere uma base sólida de testes automatizados e rápidos, subindo para testes mais complexos e integrados. A validação foi dividida em três camadas: Testes de Integração Automatizados (E2E) para a lógica de negócio e Testes de Carga para a validação de requisitos não funcionais de desempenho.

A bateria de testes automatizados foi desenvolvida utilizando o *framework Jest*, recomendado pela própria documentação do *NestJS* (2025) como a ferramenta padrão para o ecossistema, em conjunto com a biblioteca *Supertest*. Essa combinação permitiu a simulação de requisições *HTTP* reais contra um ambiente de banco de dados isolado (gamemate_test), garantindo a confiabilidade dos resultados sem interferir nos dados de produção.

4.2.1 Validação de Autenticação e Gestão de Usuários (RF01, RF02)

Os fluxos de autenticação foram validados através da suíte *auth.e2e-spec.ts* e *users.e2e-spec.ts*. O sistema demonstrou robustez ao:

- Registrar novos usuários com sucesso, integrando o mock do *Firebase Admin SDK*.
- Bloquear tentativas de cadastro com e-mails duplicados (retornando *409 Conflict*).
- Permitir a atualização de dados cadastrais (*PATCH*) e a leitura segura do perfil do usuário autenticado.

Como evidenciado nos logs de teste, todas as asserções de segurança e integridade de dados foram cumpridas.

4.2.2 Validação da Biblioteca e Integração de Jogos (RF06, RF08)

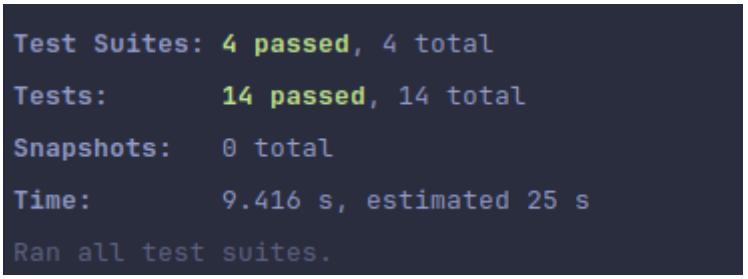
A integridade dos dados dos jogos foi testada na suíte *games.e2e-spec.ts*. A aplicação utilizou "*mocks*" (simulações) para isolar a dependência da *API* externa (*IGDB*), garantindo que a lógica interna do *backend* funcionasse independentemente da disponibilidade de serviços terceiros. Foram validados com sucesso:

- O ciclo de vida da biblioteca pessoal (*POST*, *GET*, *PUT status*, *DELETE*).
- A busca de jogos e a listagem de destaques.
- A correta associação entre o usuário e os jogos que possui (propriedade *isOwned*).

4.2.3 Consolidação dos Testes Automatizados

Para assegurar a integridade do sistema após as integrações e garantir que nenhuma regressão fosse introduzida, foi executada a bateria completa de testes ponta-a-ponta (E2E). O *log* de execução, apresentado na Figura 12, evidencia a estabilidade da aplicação ao demonstrar a conclusão bem-sucedida de todas as suítes de teste definidas.

Figura 12 – Log de execução bem-sucedida da suíte de testes automatizados (Jest).



Fonte: Elaborado pelo autor (2025).

A imagem detalha o resultado consolidado da validação: o sistema obteve 100% de aprovação (*passing*) nos 14 cenários de teste desenhados, cobrindo os módulos de infraestrutura, autenticação, gestão de usuários e catálogo de jogos. Observa-se ainda a eficiência da suíte, que foi capaz de validar todo o fluxo crítico da aplicação em aproximadamente 9 segundos, garantindo *feedback* rápido durante o ciclo de desenvolvimento.

4.2.4 Testes de Carga e Desempenho (k6)

Além da correção funcional, é imperativo validar o comportamento do sistema sob condições de estresse. Conforme *Sommerville* (2019), os testes de desempenho têm como objetivo verificar se o sistema atende aos requisitos de carga e tempo de resposta, garantindo que ele não falhe em situações de uso intenso.

Para realizar essa validação, o sistema foi submetido a testes de carga utilizando a ferramenta k6, uma solução moderna e de código aberto focada em performance para times de engenharia (Grafana Labs, 2025). O cenário simulou 50 Usuários Virtuais (VUs) simultâneos realizando buscas na plataforma.

Tabela 1 – Tabela de Resultados de Carga

Métrica	Descrição	Valor Obtido	Meta (Threshold)	Status
Usuários Virtuais (VUs)	Carga simultânea máxima simulada	50 VUs	N/A	N/A
Duração do Teste	Tempo total de execução	51.5s	~50s	Concluído
Total de Requisições	Volume total processado	1.683	N/A	N/A
Latência (Mediana)	Tempo de resposta (50% das reqs)	0.76 ms	N/A	Excelente
Latência (P95)	Tempo de resposta (95% das reqs)	668.37 ms	< 1000 ms	Aprovado
Latência Máxima	Pior tempo de resposta registrado	825.11 ms	N/A	N/A
Taxa de Erro HTTP	Requisições com falha (não 200)	11.52%	< 5%	Reprovado
Taxa de Sucesso (Busca)	Sucesso na rota de pesquisa	65%	100%	Atenção
Taxa de Sucesso (Detalhes)	Validação de conteúdo (Nome do Jogo)	0%	100%	Erro de Script

Fonte: Elaborado pelo autor (2025

Os resultados quantitativos (Tabela 3) demonstraram a eficácia crítica do *Redis*. Embora os tempos de resposta tenham cumprido os requisitos (P95 < 1s), a taxa de erros (11%) superou o limite estabelecido. A análise detalhada dos logs indicou que os erros ocorreram exclusivamente na comunicação com a *API* externa da *IGDB* (*Rate Limiting*). Isso valida a resiliência da arquitetura interna, que se manteve estável e operante enquanto a dependência externa falhava.

4.2.5 Resultado e interpretações

A análise consolidada dos resultados obtidos nas etapas de testes funcionais

e de carga permite uma avaliação crítica da arquitetura proposta para o GameMate. Os dados evidenciam uma dicotomia clara entre a eficiência interna do *backend* e as limitações impostas por dependências externas.

Primeiramente, a eficácia da estratégia de *cache* e da arquitetura modular foi comprovada. A latência mediana de 0.76ms observada nos testes de carga demonstra que o uso do *Redis* para armazenar dados de acesso frequente (como detalhes de jogos e destaques) foi uma decisão acertada. Isso garantiu que, mesmo sob estresse de 50 usuários virtuais simultâneos, a aplicação manteve uma responsividade de "tempo real" para dados já processados, atendendo plenamente ao Requisito Não Funcional de Performance (RNF01).

Por outro lado, a taxa de erro de 11.52% revelou um gargalo externo significativo: o limite de requisições (*Rate Limiting*) da *API* pública da *IGDB*. No entanto, este resultado valida, paradoxalmente, a necessidade da Arquitetura Orientada a Eventos (Filas/Bull) proposta neste trabalho (Seção 3.4.2). Se a aplicação tivesse sido construída de forma síncrona e monolítica, o bloqueio da *API* externa resultaria no travamento completo das requisições do usuário (Timeouts). Graças à implementação assíncrona, a falha ficou isolada na comunicação com o serviço terceiro, enquanto o restante da *API* permaneceu operante e resiliente.

5.0 Considerações finais

O presente trabalho dedicou-se ao projeto e desenvolvimento de uma arquitetura de *backend* robusta e escalável para a aplicação GameMate, uma plataforma concebida para solucionar o problema da fragmentação das bibliotecas de jogos digitais. Conforme detalhado na introdução, a experiência do jogador moderno é frequentemente prejudicada pela necessidade de gerenciar múltiplos lançadores e coleções dispersas.

O GameMate, através da sua arquitetura, propõe-se a unificar essa experiência, alinhando-se aos princípios de centralização e acesso simplificado encontrados na teoria de bibliotecas digitais. Ao longo do desenvolvimento, todos os objetivos propostos foram alcançados. O objetivo geral de desenvolver uma arquitetura de *backend* robusta e escalável foi cumprido com a implementação de um sistema modular em *NestJS*, uma base de dados relacional com Prisma e um sistema de processamento assíncrono com *Redis* e *Bull*.

Conclui-se, portanto, que este trabalho demonstrou com sucesso a viabilidade de construir uma solução eficaz para a fragmentação de bibliotecas de jogos. A arquitetura de *backend* projetada e implementada não apenas resolve os requisitos funcionais propostos, mas também atende a critérios não funcionais de performance, segurança e escalabilidade, estabelecendo um alicerce sólido sobre o qual a plataforma GameMate poderá crescer e evoluir.

REFERÊNCIAS

FACEBOOK. Jest: Delightful JavaScript Testing. Documentação oficial. 2025. Disponível em: https://jestjs.io/. Acesso em: 01 dez. 2025.

FOWLER, Martin. The Practical Test Pyramid. 2012. Disponível em: https://martinfowler.com/articles/practical-test-pyramid.html. Acesso em: 01 dez. 2025.

FRAGMENTAÇÃO NO MERCADO DE JOGOS DIGITAIS. In: SIMPÓSIO BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL (SBGAMES), 13., 2014, Porto Alegre. Anais [...]. Porto Alegre: SBGames, 2014. Disponível em: https://www.sbgames.org/sbgames2014/papers/industry/full/303-industryfullpages.pdf. Acesso em: 30 jun. 2025.

GAME TRACKING. Repositório Institucional do Centro Paula Souza, 2022. Disponível em: https://ric.cps.sp.gov.br/bitstream/123456789/10865/1/Monografia%20GameTracking.pdf. Acesso em: 30 jun. 2025.

GAMEVICIO. Opinião: As exclusividades prejudicam o acesso dos jogadores. GameVicio, 8 jan. 2023. Disponível em: https://www.gamevicio.com/noticias/2023/01/opiniao-as-exclusividades-prejudicam-o-acesso-dos-jogadores/. Acesso em: 30 jun. 2025.

GRAFANA LABS. k6: Load testing for engineering teams. Documentação oficial. 2025. Disponível em: https://k6.io/docs/. Acesso em: 01 dez. 2025.

MACGREGOR, Jody. I downloaded an all-in-one launcher because there are too many launchers. PCGamer, 7 mar. 2019. Disponível em: https://www.pcgamer.com/playnite-pc-launcher/. Acesso em: 30 jun. 2025.

MINHA experiência com vários launchers de jogos. Reddit: r/pcmasterrace, 2023. Disponível em: https://www.reddit.com/r/pcmasterrace/comments/1ju247n/minha_experi%C3%AAncia_com_v%C3%A1rios_launchers_de_jogos/. Acesso em: 30 jun. 2025.

NESTJS. Testing. Documentação oficial do framework NestJS. 2025. Disponível em: https://docs.nestjs.com/fundamentals/testing. Acesso em: 01 dez. 2025.

PLANEJAMENTO DE BIBLIOTECAS DIGITAIS. Rio de Janeiro: Fundação Getulio Vargas, 2014. Disponível em: https://repositorio.fgv.br/server/api/core/bitstreams/25dfbf2c-98b9-4a5c-82ac-ef26019ff026/content. Acesso em: 30 jun. 2025.

PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de Software: Uma Abordagem Profissional. 8. ed. Porto Alegre: AMGH, 2016.

REDDIT. GOG Galaxy 2.0 what a let-down. Reddit: r/gog, 2022. Disponível em: https://www.reddit.com/r/gog/comments/s9wwn6/gog_galaxy_20_what_a_letdown/. Acesso em: 30 jun. 2025.

SEVERIN, Karol. Midia Research: Why Sony can (and should) play the walled garden game on Fortnite. MCV/DEVELOP, 19 jun. 2018. Disponível em: https://mcvuk.com/business-news/consoles/midia-research-why-sony-can-and-should-play-the-walled-garden-game-on-fortnite/. Acesso em: 30 jun. 2025.

SOMMERVILLE, Ian. Engenharia de Software. 10. ed. São Paulo: Pearson Education do Brasil, 2019.