INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DO RIO GRANDE DO NORTE

PAULO FERNANDES VIEIRA

OmniBus: Uma Proposta de Arquitetura para Gestão de Identidade Dinâmica e Escalabilidade utilizando Java 21 e Clean Architecture

PARNAMIRIM/RN
2026

PAULO FERNANDES VIEIRA

OmniBus: Uma Proposta de Arquitetura para Gestão de Identidade Dinâmica e Escalabilidade utilizando Java 21 e Clean Architecture

Trabalho de Conclusão de Curso apresentado à Diretoria Acadêmica do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte – Campus Parnamirim, como requisito necessário à obtenção do título de Tecnólogo em sistemas para internet.

Orientador: Prof. DIEGO HENRIQUE OLIVEIRA DE SOUZA

PARNAMIRIM/RN

2026

PAULO FERNANDES VIEIRA

OmniBus: Uma Proposta de Arquitetura para Gestão de Identidade Dinâmica e Escalabilidade utilizando Java 21 e Clean Architecture

Trabalho de Conclusão de Curso apresentado à Diretoria Acadêmica do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte – Campus Parnamirim, como requisito necessário à obtenção do título de Tecnólogo em sistemas para internet.

Orientador: Prof. DIEGO HENRIQUE OLIVEIRA DE SOUZA

Aprovado em __/__/_____

BANCA EXAMINADORA

_____
Prof. DIEGO HENRIQUE OLIVEIRA DE SOUZA

Orientador – Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte

_____
Prof. JOAO MARIA ARAUJO DO NASCIMENTO

_____
Prof. FABIO AUGUSTO PROCOPIO DE PAIVA

“A imaginação é mais importante que a ciência,
porque a ciência é limitada, ao passo que
a imaginação abrange o mundo inteiro”
– Albert Einstein

Dedico este trabalho à minha mãe,

que me ajudou nos momentos que mais

precisei e me educou, conforme suas

condições.

AGRADECIMENTOS

Aos meus professores, com quem tive a honra de aprender. Gostaria de expressar minha mais profunda gratidão a todos os docentes que cruzaram meu caminho, mas especialmente àqueles que tiveram a paciência e a dedicação de guiar um aluno por vezes complicado, mas sempre sedento por conhecimento. Agradeço imensamente aos professores **Diego Oliveira (meu orientador)**, **Diego Bagatela**, **Fabio Procopio**, **Joao Maria**, **Daniel Aguiar**, **Valério Gutemberg**, **Renan Alves**, **Alvaro Hermano**, **Gustavo Fontoura**, **Glenda Michelle** e **Bruno Emerson**. Seus ensinamentos foram a base de toda a minha formação técnica e humana.

Aos pesquisadores e integrantes do **Laboratório NOCS**, agradeço por me receberem e por transformarem este espaço no lugar onde, de fato, iniciei minha jornada como um "programador de verdade". Um agradecimento especial e fraterno ao meu grande amigo **Jan Erick**, pelo apoio fundamental e pela presença constante nas horas mais difíceis; sua amizade foi um pilar nesta caminhada.

Ao **IFRN Campus Parnamirim**, instituição que me acolheu e proporcionou a infraestrutura necessária para que eu pudesse chegar ao dia de hoje. Agradeço ao atual diretor do campus, **Paulo Vitor**, por incentivar constantemente nossa participação em eventos de tecnologia, trazendo motivações e emoções que foram cruciais para o meu crescimento profissional.

À minha família, por ser o alicerce de tudo. Obrigado por terem me dado um voto de confiança quando decidi mudar de rumo, saindo de outro curso sem saber exatamente para onde ir, movido apenas pela esperança de dias melhores. Sem o apoio e o suporte de vocês, essa formatura não seria possível.

Aos meus amigos, que tornaram os dias mais leves e sempre acreditaram na minha capacidade. Um agradecimento especial ao meu grande amigo **Thiago**, que sempre me permitiu continuar quando eu já não tinha mais forças e esteve ao meu lado em todos os momentos de necessidade. A **Marcos (Marcola)**, pela preocupação constante com o meu bem-estar. A **Davi Emanuel**, **Rickson**, **Ana Clara** e **Bruno Python**, agradeço imensamente por terem tido fé em mim e depositado total confiança nos meus conhecimentos técnicos, mesmo nos momentos de incerteza. Agradeço também a **Bruna Medina**, **Amanda**, **Fernando**, **Augusto**, **Gian**, **Hugo**, **Sousa Erick** e **Gustavo**.

Em especial, dedico uma **menção honrosa** ao meu grande e saudoso amigo, **Ryan Sousa**. Estivemos juntos durante boa parte da minha graduação anterior e compartilhamos os quatro anos de IFRN no curso técnico. Sei que ele gostaria de estar aqui hoje celebrando cada conquista. Finalmente, amigo, completei o ciclo. Não terminei o curso ao seu lado, mas levei você no coração em todos os momentos.

Por fim, agradeço a todos que participaram, mesmo que brevemente, desta jornada. Ela não se encerra aqui; é apenas o começo de uma trajetória que, certamente, continuará a ser escrita com a ajuda de muitas outras mãos. Muito obrigado a todos!

.

RESUMO

Este trabalho apresenta o desenvolvimento do protótipo **OmniBus**, uma API voltada para a gestão de transportes, construída sob os preceitos da ***Clean Architecture*** e utilizando as funcionalidades modernas do **Java 21**, como ***Records***. A problemática central originou-se da observação prática no projeto **GEPAG (IFRN)**, onde se identificou que a rigidez na gestão de perfis de acesso gerava fricção no ciclo de desenvolvimento e redundância de dados. Como solução, o protótipo implementa um sistema de **Identidade Dinâmica** baseado em **Spring Security** e **JWT**, permitindo que um único usuário alterne entre contextos de "Passageiro" e "Empresa" de forma segura e transparente. A metodologia envolveu a integração de uma infraestrutura integra composta por **PostgreSQL** para persistência relacional, **MinIO** para armazenamento de objetos e **Hibernate Envers** para auditoria automática de dados. A qualidade e a saúde do software foram asseguradas através de testes automatizados com **JUnit** e **Mockito**, análise estática via **SonarQube** e monitoramento de falhas em tempo real com **Sentry**. Os resultados demonstram que a separação clara de responsabilidades e o uso de padrões de projeto modernos garantem a escalabilidade, a observabilidade e a manutenibilidade do software, provendo uma fundação técnica sólida para a evolução de sistemas corporativos complexos.

Palavras-chave: Java 21. Clean Architecture. Identidade Dinâmica. Spring Boot 3. Observabilidade.

O código-fonte e a documentação completa do protótipo OmniBus estão disponíveis em: https://github.com/PaulinhoVieira/omnibus-api

SUMÁRIO

1. INTRODUÇÃO

No cenário contemporâneo do desenvolvimento de software, a celeridade na entrega frequentemente entra em conflito com a sustentabilidade do código. Sistemas corporativos íntegros demandam não apenas funcionalidade imediata, mas atributos de qualidade como manutenibilidade e confiabilidade. Segundo **Martin (2019)**, a arquitetura de um software não deve ser ditada por ferramentas ou frameworks, mas sim pelas regras de negócio, garantindo que o sistema seja "independente de detalhes" e fácil de evoluir.

A problemática central reside no **acoplamento excessivo** entre a lógica de negócio e as tecnologias externas (bancos de dados, interfaces de usuário e APIs), o que compromete a visibilidade sobre falhas e cria estruturas rígidas. Quando as dependências não são devidamente invertidas, a evolução do sistema torna-se um processo de alto risco e elevado custo operacional. Nesse sentido, a adoção de padrões arquiteturais sólidos, como a **Clean Architecture**, aliada ao uso de linguagens em versões LTS (*Long Term Support*), como o Java 21, surge como uma estratégia fundamental para garantir o desacoplamento e a longevidade do produto tecnológico.

1.1 Problematização

A motivação para este trabalho originou-se de desafios práticos observados no desenvolvimento de soluções de software voltadas à gestão logística. Inicialmente, o projeto **OmniBus** foi concebido como uma resposta à precariedade dos sistemas de bilhetagem rodoviária, onde a necessidade de deslocamento físico dos passageiros aos terminais para a compra presencial de passagens evidenciou uma lacuna de digitalização e eficiência no setor. A proposta inicial visava unificar a venda de bilhetes em uma plataforma digital, reduzindo a carga operacional das rodoviárias e oferecendo autonomia para empresas e usuários.

Entretanto, ao transpor essa necessidade de negócio para o ambiente de desenvolvimento, surgiram barreiras técnicas que vão além da funcionalidade básica. Durante a participação do autor no projeto de extensão **GEPAG (Arroz da Gente)**, no âmbito do IFRN, observou-se que modelos tradicionais de arquitetura, muitas vezes de natureza monolítica, tratam perfis de acesso de forma estática e rígida. No contexto do OmniBus, essa rigidez manifestava-se na dificuldade de gerenciar múltiplos papéis (como "Passageiro" e "Empresa") dentro de uma mesma conta, obrigando desenvolvedores e

homologadores a criarem e gerenciarem diversas credenciais para testar diferentes níveis de permissão.

Este cenário resulta em um aumento significativo da dívida técnica e na perda de produtividade, colidindo com as exigências atuais do mercado de software, que demanda eficiência, qualidade e rapidez na entrega (time-to-market). Diante dessas experiências, percebeu-se que o problema não residia apenas na funcionalidade de venda de passagens, mas na falta de uma estrutura que permitisse a evolução sustentável do código. Assim, o projeto evoluiu de um simples sistema de vendas para a proposição de uma arquitetura de referência que utiliza o conceito de **Identidade Dinâmica**, isolando a lógica de negócio das ferramentas externas e flexibilizando a operação do sistema.

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver e documentar uma arquitetura de referência para sistemas *back-end*, denominada **OmniBus**, fundamentada nos princípios de *Clean Architecture* e nas funcionalidades modernas do **Java 21**.

Objetivos Específicos:

1. **Projetar e implementar** uma arquitetura modular baseada em camadas utilizando o framework Spring Boot 3;
2. **Desenvolver** um sistema de autenticação via JWT que suporte a alternância de perfis (Identidade Dinâmica) em uma única conta de usuário;
3. **Estabelecer uma infraestrutura de suporte** utilizando MinIO para armazenamento de objetos e Hibernate Envers para rastreabilidade histórica dos dados;
4. **Validar a integridade** do sistema através de uma pirâmide de testes automatizados com JUnit e Mockito;
5. **Aplicar ferramentas** de análise estática e observabilidade em tempo real, como SonarQube e Sentry, para assegurar a qualidade do software.

1.3 Justificativa

A relevância deste trabalho manifesta-se em três esferas principais:

- **Esfera Acadêmica:** Ao explorar o estado da arte do Java 21 e os preceitos da *Clean Architecture*, servindo de guia para o desenvolvimento de softwares acadêmicos maduros;
- **Esfera Profissional:** Ao propor uma solução técnica para a gestão complexa de múltiplos perfis, problema identificado em cenários reais de desenvolvimento (Projeto GEPAG);
- **Esfera Social:** Ao fomentar a criação de sistemas mais estáveis, seguros e menos propensos a falhas críticas, garantindo uma melhor experiência para o usuário final.

1.4 Método de Pesquisa

A pesquisa adotará o método de Desenvolvimento Experimental e Pesquisa Bibliográfica. A base teórica será construída sobre as obras de Robert C. Martin (*Clean Architecture*) e os princípios SOLID. Na fase prática, será desenvolvido um protótipo funcional onde a eficiência da arquitetura será validada por meio de relatórios de cobertura de código no SonarQube e o rastreamento de exceções em tempo real via Sentry.

1.5 Estrutura do Trabalho

O presente trabalho está organizado da seguinte forma:

- **Capítulo 1 (Introdução):** Apresenta o tema, os objetivos e a metodologia.
- **Capítulo 2 (Referencial Teórico):** Discorre sobre Java 21, *Clean Architecture*, Segurança e observabilidade.
- **Capítulo 3 (Metodologia e Tecnologias):** Detalha as ferramentas, o ambiente Docker e os procedimentos de implementação.
- **Capítulo 4 (Desenvolvimento do Protótipo):** Detalha a modelagem do sistema, a lógica de Multi-Perfis e a estratégia de qualidade.
- **Capítulo 5 (Conclusão):** Sintetiza as considerações finais e sugestões de trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

2.1 A Refatoração como Paradigma de Design Contínuo

O desenvolvimento de sistemas íntegros em Java e Spring Boot exige uma mudança na forma como o design de software é concebido. Historicamente, acreditava-se que o planejamento estrutural deveria ser estático e finalizado antes da codificação. No entanto, Fowler (1999) argumenta que essa abordagem leva inevitavelmente à decadência do sistema. A refatoração surge, portanto, como uma prática disciplinada de melhorar a estrutura interna sem alterar o comportamento externo. Segundo o autor:

> "A refatoração é o oposto dessa prática [hacking]. Com ela, podemos partir de um design ruim, até mesmo caótico, e transformá-lo em um código bem estruturado. [...] O resultado dessa interação é um programa cujo design permanece bom enquanto o desenvolvimento continua." (FOWLER, 1999, p. 14).

Para este trabalho, a refatoração é o pilar que permite que um projeto com tecnologias complexas (MinIO, Docker, Postgres) não se torne obsoleto, garantindo a "integridade do sistema" através de melhorias constantes.

2.2 Clean Code e a Comunicação com o Desenvolvedor

Complementando a refatoração, o conceito de *Clean Code* estabelece que o código deve ser escrito para humanos. Martin (2009) reforça que a clareza é a característica mais importante de um sistema. Conforme Fowler (1999, p. 31), "Qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que humanos entendam." No contexto do Java 21, o uso de recursos como *Records* e *Pattern Matching* são formas de aplicar esse princípio, reduzindo a carga cognitiva e tornando as regras de negócio autoexplicativas.

2.3 Arquitetura Limpa e Qualidade Mensurável

Para que o sistema seja sustentável, a lógica de negócio deve ser isolada de detalhes de implementação (MARTIN, 2017). A integridade dessa arquitetura no projeto OmniBus é garantida por:

- **Testes Automatizados:** Validam se a refatoração não alterou o comportamento externo (BECK, 2003).
- **SonarQube:** Atua como revisor automático de *Code Smells* e métricas de qualidade.
- **Sentry:** Garante a observabilidade em tempo real, identificando falhas em tempo de execução.

2.4 Padronização e Produtividade em Equipes (GEPAG)

A produtividade depende da eficiência da comunicação. Conforme observado na prática no projeto **GEPAG (Arroz da Gente)**, a ausência de padrões gera um custo de comunicação elevado. A adoção de Contratos de Interface (Swagger) e padrões de DTOs permite que o desenvolvimento ocorra de forma assíncrona e previsível, reduzindo reuniões e focando em regras de negócio.

2.5 Persistência de Dados e Auditoria (JPA e Envers)

A camada de persistência utiliza o **Spring Data JPA**, que resolve o "descompasso de impedância" entre objetos e tabelas relacionais (KING; BAUER, 2015). Para a rastreabilidade, o **Hibernate Envers** automatiza a auditoria. Como aponta a documentação técnica, "o Envers fornece uma solução de auditoria completa para entidades de domínio, permitindo recuperar versões históricas dos dados" (HIBERNATE, 2026), garantindo a segurança exigida em sistemas de gestão através das tabelas de histórico (`_aud`).

2.6 Segurança e Gestão de Identidade (Spring Security e JWT)

Para garantir a integridade da identidade dinâmica (multi-perfil), utiliza-se o **Spring Security**. Walls (2019) afirma que este *framework* fornece uma infraestrutura íntegra para autenticação e controle de acesso. A integração com o padrão **JSON Web Token (JWT)** permite que o estado da autenticação seja transmitido de forma segura e sem estado (*stateless*), validando a alternância entre perfis (Passageiro/Empresa) a cada requisição sem sobrecarregar o servidor.

3. METODOLOGIA

A presente pesquisa adota o método de **Desenvolvimento Experimental**, pautado na criação de um protótipo funcional para validação de hipóteses técnicas em engenharia de software. O estudo fundamenta-se em uma abordagem de **Pesquisa-Ação**, uma vez que o objeto de estudo nasce de lacunas reais de produtividade e manutenção identificadas durante a trajetória acadêmica e profissional do autor no IFRN (cursos de TSI e atividades no NOCS), especificamente no projeto "Arroz da Gente" (GEPAG).

3.1 Identificação do Problema e Requisitos

A fase inicial consistiu no levantamento de gargalos em sistemas legados e monólitos de alta complexidade. Identificou-se que a ausência de padrões arquiteturais e o uso de funções extensas ("código macarrônico") elevavam a carga cognitiva da equipe, gerando dependência excessiva de comunicação verbal entre desenvolvedores *back-end* e *front-end*. Como requisito técnico principal, definiu-se a necessidade de gerir **múltiplos perfis de acesso** em uma única conta de usuário, otimizando a integridade do sistema sem comprometer a experiência de uso.

3.2 Definição da Arquitetura e Stack Tecnológico

Para solucionar os problemas identificados, optou-se pelo desenvolvimento do protótipo OmniBus. A organização do projeto segue os preceitos da Arquitetura Limpa, conforme pode ser visualizado na **Figura 1**, que detalha a estrutura de pastas e a separação de responsabilidades no repositório do projeto.

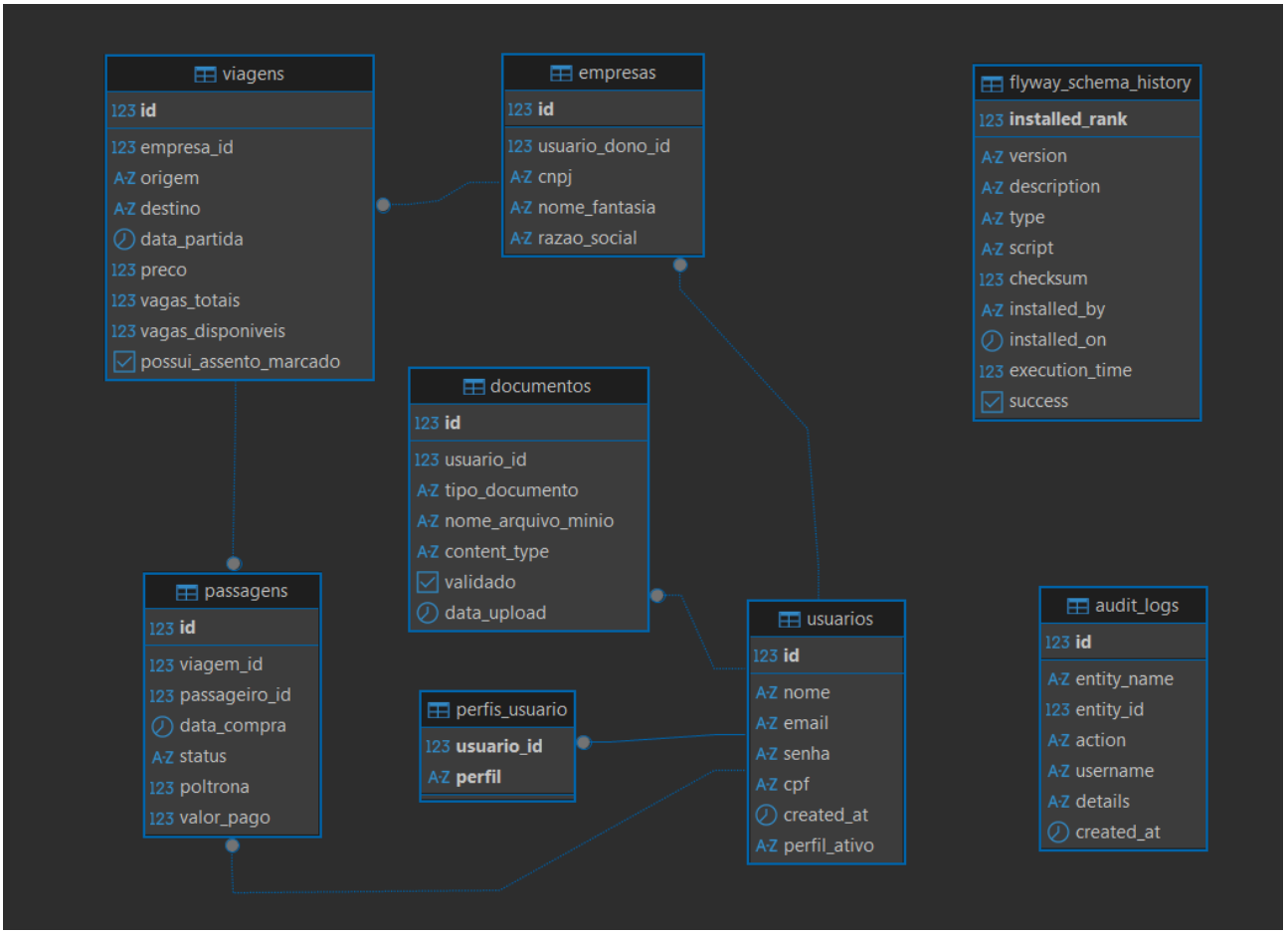


Figura 1 – Diagrama e arquitetura do protótipo. Fonte: Autor (2026).

A stack tecnológica utilizada inclui:

- **Linguagem e Framework:** Java 21 (utilizando *Records* para imutabilidade) e Spring Boot 3.
- **Segurança:** Implementação de RBAC (*Role-Based Access Control*) via Spring Security e tokens JWT, permitindo a gestão dinâmica de permissões.
- **Persistência e Infraestrutura:** PostgreSQL e MinIO orquestrados via Docker. A organização desses serviços pode ser observada na **Figura 2**, que apresenta o arquivo de configuração de ambiente.

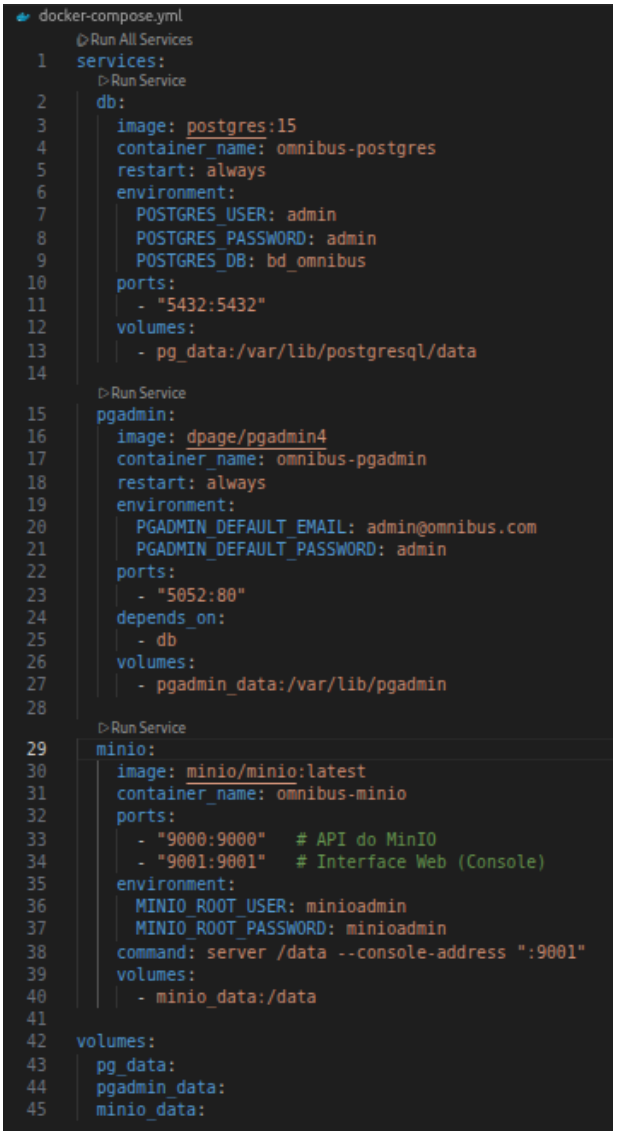


Figura 2 – Configuração de infraestrutura via Docker Compose. Fonte: Autor (2026).

3.3 Procedimentos de Implementação e Clean Code

A implementação foi guiada pelos princípios da **Clean Architecture**, visando a redução do atrito no ciclo de desenvolvimento. Os procedimentos adotados incluíram:

- **Decomposição de Funções:** Fragmentação de lógicas complexas em subfunções especialistas com nomenclaturas semânticas, visando um código "auto-documentável" que dispense comentários excessivos.

- **Padronização de Contratos (API):** Utilização do Swagger (OpenAPI 3) como ferramenta de contrato entre camadas como pode ser visto na **Figura 3**, permitindo que a equipe de *front-end* atue de forma assíncrona e independente.

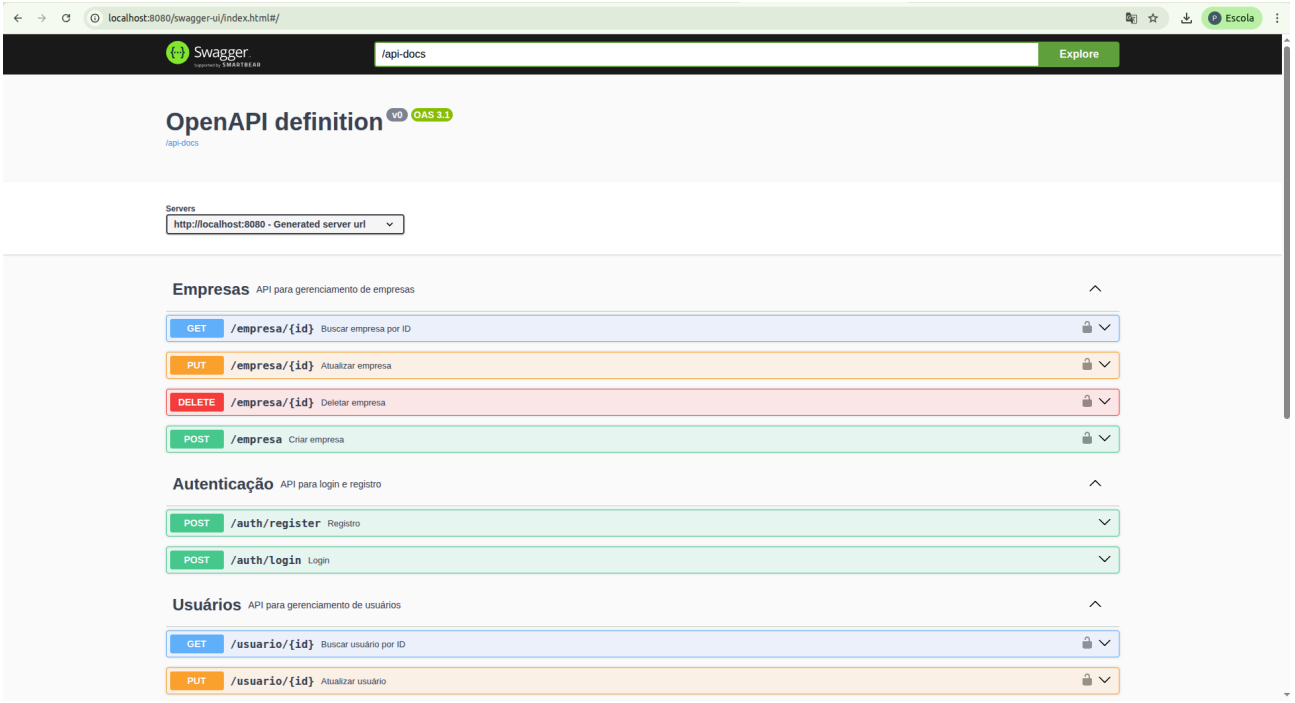


Figura 3 – Interface do Swagger UI apresentando a documentação e os contratos da API.

- **Refatoração como Paradigma:** Aplicação constante de melhorias na estrutura interna do código para evitar a degradação do sistema frente às pressões de prazo e complexidade.

A eficácia da metodologia e do protótipo será validada por meio de indicadores quantitativos e qualitativos:

- **Análise Estática via SonarQube:** Mensuração de *Code Smells*, dívida técnica e cobertura de testes automatizados (JUnit/Mockito). Conforme apresentado na **Figura 4**, a ferramenta permite visualizar o estado da saúde do código em tempo real, garantindo que os padrões de qualidade sejam mantidos durante o processo de refatoração.

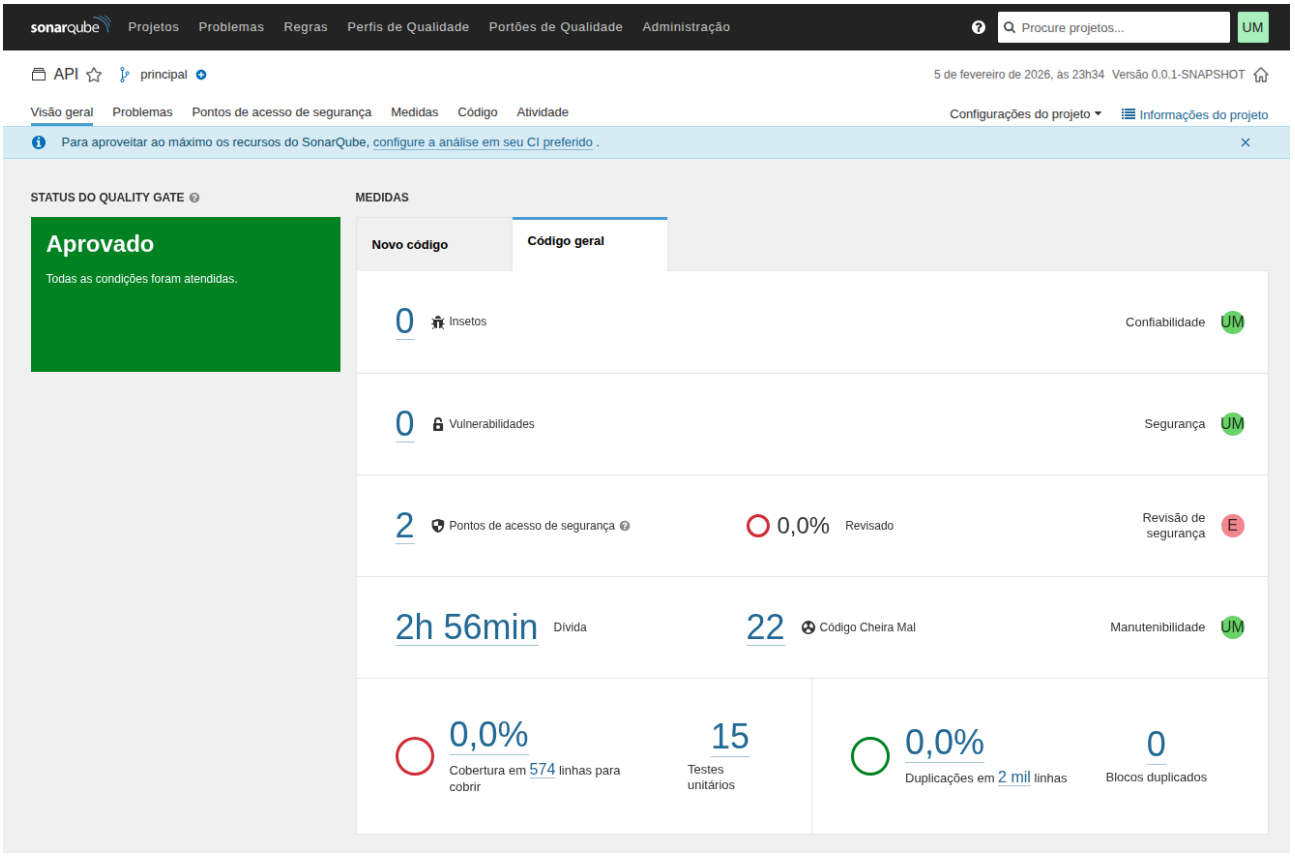


Figura 4 – Dashboard de métricas de qualidade e análise estática via SonarQube. Fonte: Autor (2026).

- **Observabilidade via Sentry:** Monitoramento proativo de exceções e rastreamento de fluxos em tempo real, validando a estabilidade da lógica de múltiplos perfis.
- **Eficiência de Comunicação:** Avaliação da clareza das rotas e DTOs como redutores da necessidade de alinhamentos técnicos síncronos entre desenvolvedores.

4. DESENVOLVIMENTO

4.1 Contextualização e Motivação do Protótipo

O desenvolvimento do protótipo **OmniBus** não é fruto de uma necessidade puramente acadêmica, mas de observações práticas realizadas durante a trajetória do autor no ecossistema de desenvolvimento do IFRN, com destaque para o projeto "Arroz da Gente" (GEPAG). A principal motivação foi encontrar um equilíbrio entre a celeridade das entregas e a sustentabilidade técnica a longo prazo.

Em cenários de desenvolvimento acelerado, a ausência de padrões rigorosos frequentemente resulta no acúmulo de dívida técnica. Este capítulo detalha como a aplicação dos princípios de *Clean Architecture* e o uso de recursos modernos do Java 21 foram empregados para criar uma base de código íntegra, capaz de mitigar gargalos de comunicação e facilitar a implementação de funcionalidades complexas, como a gestão dinâmica de múltiplos perfis de usuário.

4.1.1 O Fator Humano e a Flexibilidade dos Padrões

É fundamental ressaltar que as práticas de engenharia aqui propostas não são dogmas absolutos. O desenvolvimento de software é uma disciplina sociotécnica, onde a eficiência de um método depende diretamente da cooperação e do contexto da equipe. No projeto GEPAG, observou-se que a agilidade extrema pode levar à "erosão arquitetural" quando os padrões são ignorados. Um exemplo prático é a inconsistência na nomenclatura de DTOs (como a dúvida entre `Empresario` ou `EmpresarioNome`), que gera um efeito cascata de retrabalho para o desenvolvedor *front-end*.

4.1.2 O Desafio da Padronização Linguística e Técnica

A gestão do idioma no código também foi um ponto de atenção. Embora o inglês seja o padrão global, projetos locais frequentemente adotam o português. O protótipo demonstra que, independentemente da língua escolhida, a **consistência** é a chave. A infraestrutura proposta assegura que os contratos de interface (API) permaneçam íntegros, servindo como um guia que reduz a necessidade de alinhamentos verbais constantes, assim como demonstrado na **Figura 5**.

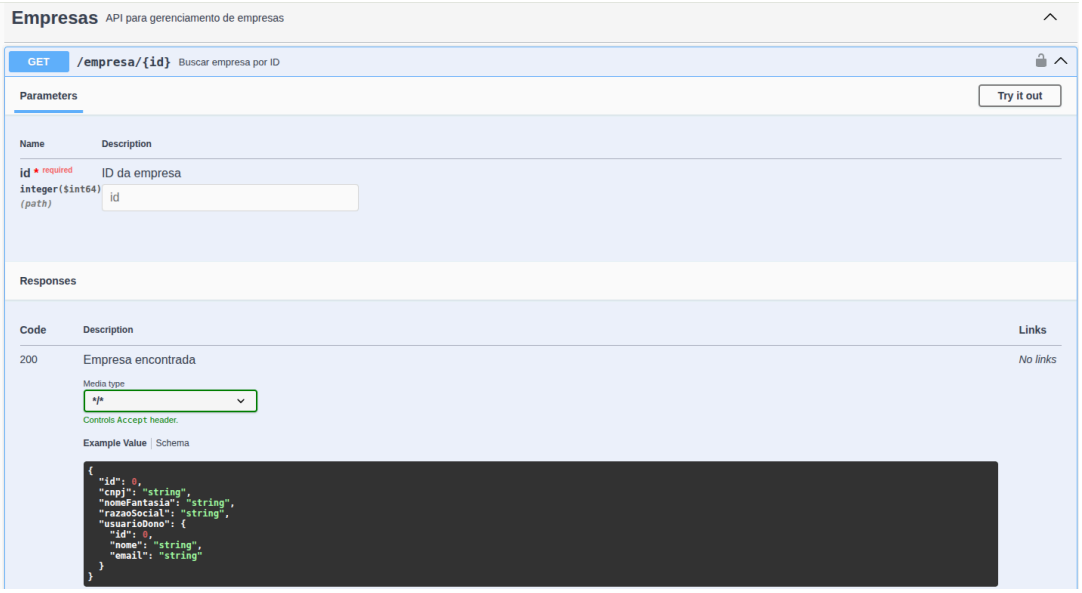


Figura 5 – Interface de documentação da API (Swagger UI). Fonte: Autor (2026).

4.2 Arquitetura e Estrutura do Sistema

A OmniBus API foi organizada em camadas para garantir a separação de responsabilidades de acordo com a **Figura 6**. A estrutura de pacotes (como `domain`, `service`, `repository` e `controller`) reflete essa preocupação.

- **Domínio e Identidade:** Optou-se pelo uso de chaves primárias do tipo **Long** com auto-incremento. Esta decisão visa otimizar a performance de indexação e facilitar a rastreabilidade manual durante os testes, comparado ao uso de UUIDs.
- **Identidade Dinâmica:** Utilizou-se a anotação `@ElementCollection` para permitir que um único usuário possua múltiplos papéis (ex: PASSAGEIRO e EMPRESA) de forma nativa.

```
omnibus-api/
│
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── br/com/vendas/passagem/omnibus/
│   │   │       ├── OmnibusApiApplication.java          # Classe principal da aplicação
│   │   │       ├── annotation/                         # Anotações customizadas
│   │   │       ├── config/                             # Configurações da aplicação
│   │   │       ├── controller/                         # Controllers REST
│   │   │       ├── domain/                             # Entidades do domínio
│   │   │       ├── dto/                                # Objetos de Transferência de Dados
│   │   │       ├── exception/                          # Tratamento de exceções
│   │   │       ├── repository/                         # Camada de persistência
│   │   │       └── service/                            # Lógica de negócio
│   │   │
│   │   └── resources/
│   │       ├── application.properties                  # Configurações da aplicação
│   │       ├── db/migration/                           # Scripts de migração Flyway
│   │       ├── static/                                 # Arquivos estáticos
│   │       └── templates/                              # Templates (se aplicável)
│   │
│   └── test/
│       └── java/                                       # Testes unitários e integração
│
├── target/                                             # Artefatos compilados (não versionado)
├── docker-compose.yml                                  # Configuração Docker
├── pom.xml                                             # Configuração Maven
├── HELP.md                                             # Documentação de ajuda
├── EXEMPLOS_TESTES_API.md                             # Exemplos de testes da API
└── ARQUITETURA.md                                      # Este documento
```

Figura 6 – Organização de pacotes seguindo a Arquitetura em Camadas. Fonte: Autor (2026).

4.3 Gestão de Contexto e Segurança (JWT)

O diferencial técnico reside no ato da autenticação contextual. Ao realizar o login, o cliente envia as credenciais e o **perfil desejado** para aquela sessão (`perfilDesejado`). O sistema valida se o usuário possui tal permissão e, se for o caso de um perfil de "Empresa", verifica o vínculo obrigatório com uma transportadora. O token JWT resultante contém apenas a autoridade específica escolhida, aplicando o **Princípio do Privilégio Mínimo**. A **Figura 7** demonstra a base de como ficou as rotas de login e registrar, juntamente com a **Figura 8** que demonstra o fluxo de login.

```java
@Controller
@RequestMapping("/auth")
public class AuthenticationController {

    private final UsuarioRepository usuarioRepository;

    private final AuthenticationManager authenticationManager;
    private final UsuarioService usuarioService;
    private final TokenService tokenService;

    public AuthenticationController(AuthenticationManager authenticationManager, UsuarioService usuarioService,
                                    UsuarioRepository usuarioRepository, TokenService tokenService) {
        this.authenticationManager = authenticationManager;
        this.usuarioService = usuarioService;
        this.usuarioRepository = usuarioRepository;
        this.tokenService = tokenService;
    }

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody AuthenticationDTO authRequest) {
        // Buscar usuário para validar perfil (com empresas se necessário)
        Usuario usuario = usuarioRepository.findByEmailWithEmpresas(authRequest.email())
                .orElseThrow(() -> new BadCredentialsException(msg: "Credenciais inválidas"));

        // Validar se o usuário possui o perfil solicitado
        if (!usuario.possuiPerfil(authRequest.perfilDesejado())) {
            throw new BadCredentialsException("Você não possui permissão para acessar como " + authRequest.perfilDesejado());
        }

        // Se o perfil for EMPRESA, validar se tem empresa associada
        if (authRequest.perfilDesejado() == TipoPerfil.EMPRESA && (usuario.getEmpresas() == null || usuario.getEmpresas().isEmpty())) {
            throw new BadCredentialsException("Você não possui empresa cadastrada" + authRequest.perfilDesejado());
        }

        // Autenticar
        var usernamePassword = new UsernamePasswordAuthenticationToken(authRequest.email(), authRequest.senha());
        var authentication = authenticationManager.authenticate(usernamePassword);

        String token = tokenService.gerarToken((Usuario) authentication.getPrincipal(), authRequest.perfilDesejado());

        return ResponseEntity.ok(token);
    }

    @PostMapping("/register")
    public ResponseEntity<Void> register(@RequestBody UsuarioRequestDTO request) {
        if(usuarioRepository.findByEmail(request.email()).isPresent()) {
            throw new BadCredentialsException("Usuário já existe com o email: " + request.email());
        }

        // Service vai criptografar a senha, adicionar perfil PASSAGEIRO e garantir a persistência
        usuarioService.criarUser(request);
        return ResponseEntity.ok().build();
    }

}
```

Figura 7 – Lógica de validação de credenciais e perfil contextual no Controller. Fonte: Autor (2026).

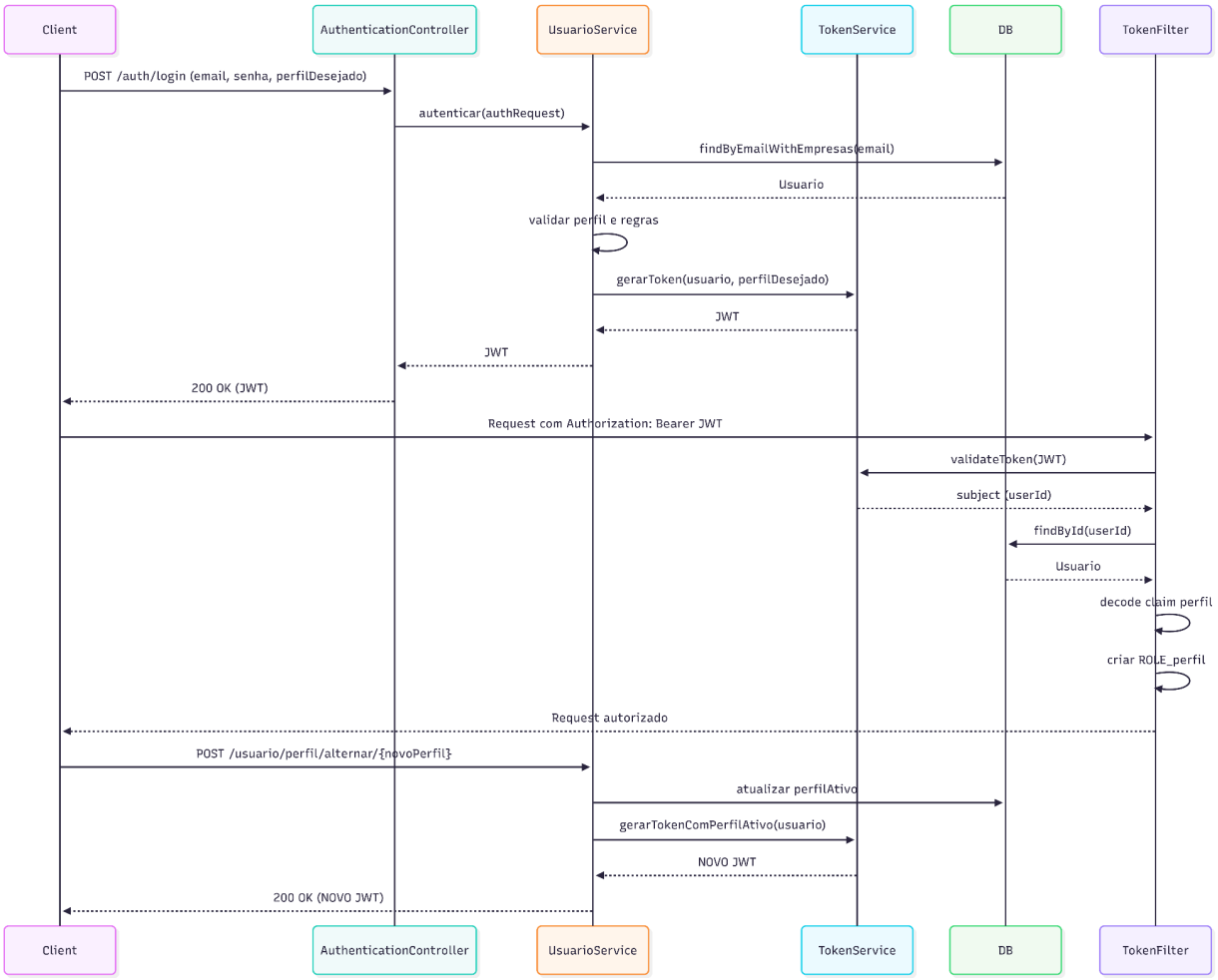


Figura 8 – diagrama de sequência mostrando fluxo do login com multi perfis. Fonte: Autor (2026).

Este diagrama de sequência descreve o fluxo de autenticação e alternância de perfis em um sistema com múltiplas permissões por usuário. No login, o cliente envia e-mail, senha e o perfil desejado; o serviço valida credenciais, verifica se o usuário possui o perfil solicitado (e regras adicionais, como existência de empresa) e então emite um **JWT** contendo o claim perfil. Em cada requisição subsequente, o filtro de segurança valida o token, extrai o perfil e monta a autoridade **ROLE_<perfil>** para autorizar rotas. Quando o usuário solicita a troca de perfil, a aplicação atualiza o perfil ativo no banco e retorna um novo **JWT** com o claim atualizado. Assim, somente uma permissão fica efetivamente ativa por token, garantindo coerência entre perfil escolhido e acesso às rotas protegidas.

4.4 Infraestrutura, Auditoria e Qualidade

O sistema integra-se ao **MinIO** para armazenamento de documentos, evitando o armazenamento de binários no banco de dados. Para a transparência dos dados, utilizou-se o Hibernate Envers com AOP (*Aspect Oriented Programming*) para gerar logs de auditoria automáticos como pode ser analisado na **Figura 9.**

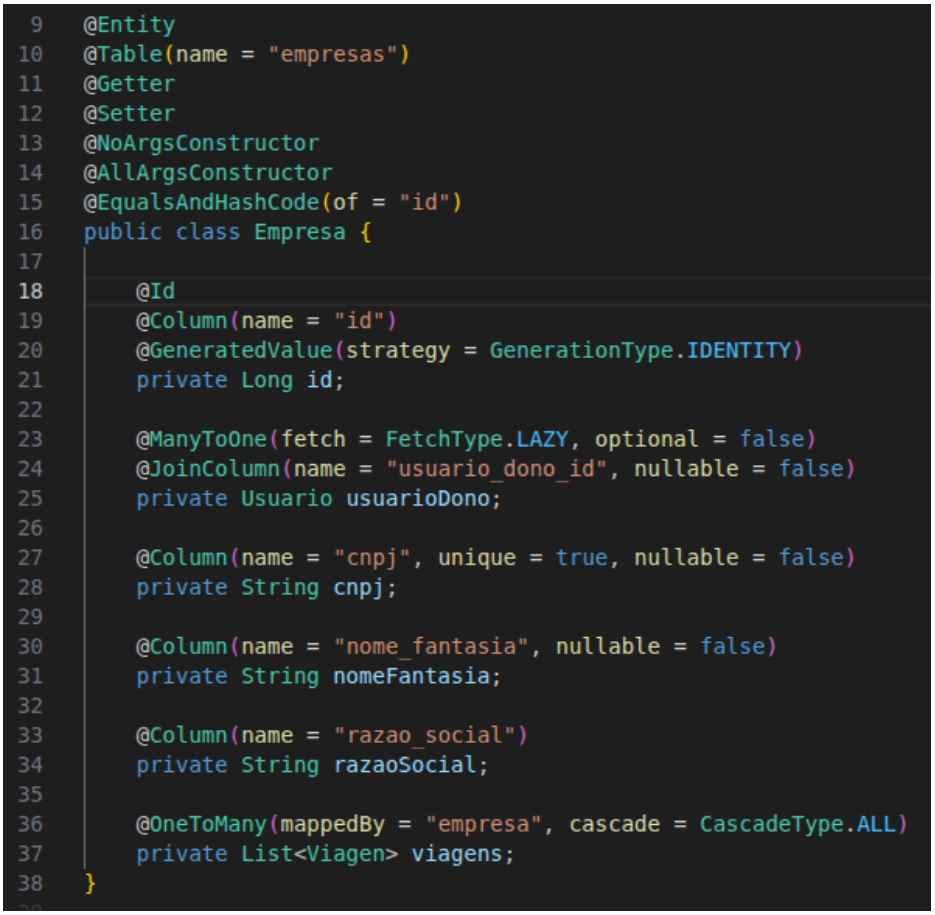
```sql
-- Tabela para registrar logs de auditoria gerais do sistema
CREATE TABLE IF NOT EXISTS audit_logs (
    id BIGSERIAL PRIMARY KEY,
    entity_name VARCHAR(150) NOT NULL,
    entity_id BIGINT NULL,
    action VARCHAR(50) NOT NULL,
    username VARCHAR(150) NULL,
    details TEXT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

-- Índices para melhorar performance nas consultas
CREATE INDEX IF NOT EXISTS idx_audit_logs_entity ON audit_logs (entity_name, entity_id);
CREATE INDEX IF NOT EXISTS idx_audit_logs_created_at ON audit_logs (created_at);
CREATE INDEX IF NOT EXISTS idx_audit_logs_username ON audit_logs (username);
```

Figura 9 – Estrutura de tabelas de histórico geradas pelo Hibernate Envers. Fonte: Autor (2026).

4.5 Camada de Domínio e Persistência

A camada de domínio foi desenhada para ser o coração da aplicação, utilizando o mapeamento objeto-relacional (ORM) com JPA/Hibernate. Um bom exemplo de como ficou pode ser visto na **Figura 10.**



```java
@Entity
@Table(name = "empresas")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")
public class Empresa {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "usuario_dono_id", nullable = false)
    private Usuario usuarioDono;

    @Column(name = "cnpj", unique = true, nullable = false)
    private String cnpj;

    @Column(name = "nome_fantasia", nullable = false)
    private String nomeFantasia;

    @Column(name = "razao_social")
    private String razaoSocial;

    @OneToMany(mappedBy = "empresa", cascade = CascadeType.ALL)
    private List<Viagen> viagens;
}
```

Figura 10 – Mapeamento de entidades e relacionamentos no domínio. Fonte: Autor (2026)

- **Entidades e Relacionamentos:** As entidades como **`Empresa`**, **`Viagem`** e **`Passagem`** utilizam chaves primárias do tipo **`Long`**. Para otimizar as consultas, foram utilizados relacionamentos *Lazy Loading* por padrão, evitando o carregamento desnecessário de dados pesados, enquanto o *Join Fetch* é aplicado em consultas específicas no **`Repository`** para evitar o problema de performance N+1.
- **Repositórios:** A camada **`repository`** estende o **`JpaRepository`**, o que permite operações de CRUD consistentes. Foram implementadas *Derived Queries* para buscas específicas, como **`findByEmail`** e **`existsByCnpj`**, garantindo uma interface de dados limpa e tipada.

4.6 Camada de Serviço (Business Logic)

Os serviços no protótipo OmniBus não são meros repassadores de dados. Eles concentram a inteligência do sistema, garantindo que as regras de negócio sejam respeitadas antes de qualquer persistência como é demonstrado na **Figura 11**.

```java
@Transactional(readOnly = true)
public String autenticar(AuthenticationDTO authRequest) {
    // 1. Buscar usuário
    Usuario usuario = usuarioRepository.findByEmailWithEmpresas(authRequest.email())
            .orElseThrow(() -> new BadCredentialsException(msg: "Credenciais inválidas"));

    // 2. Validar se o usuário possui o perfil solicitado
    if (!usuario.possuiPerfil(authRequest.perfilDesejado())) {
        throw new BadCredentialsException("Você não possui permissão para acessar como " + authRequest.perfilDesejado());
    }

    // 3. Se o perfil for EMPRESA, validar se tem empresa associada
    if (authRequest.perfilDesejado() == TipoPerfil.EMPRESA &&
        (usuario.getEmpresas() == null || usuario.getEmpresas().isEmpty())) {
        throw new BadCredentialsException(msg: "Você não possui empresa cadastrada");
    }

    // 4. Autenticar credenciais
    var usernamePassword = new UsernamePasswordAuthenticationToken(
        authRequest.email(),
        authRequest.senha()
    );
    var authentication = authenticationManager.authenticate(usernamePassword);

    // 5. Gerar e retornar token
    return tokenService.gerarToken((Usuario) authentication.getPrincipal(), authRequest.perfilDesejado());
}
```

Figura 11 – Implementação de regras de negócio e validações na camada Service. Fonte: Autor (2026).

- **Desacoplamento:** A camada **`service`** comunica-se exclusivamente com DTOs e Repositórios. Isso garante que, caso o modelo de banco de dados mude, a API externa (Controller) permaneça estável.
- **Tratamento de Exceções de Negócio:** Em vez de retornar erros genéricos, os serviços lançam exceções customizadas como **`ResourceNotFoundException`** ou **`BusinessException`**.

4.7 Objetos de Transferência de Dados (DTOs) e Mapeamento

Para respeitar os princípios de *Clean Code* e segurança, o sistema utiliza DTOs para separar o que é recebido/enviado do que é armazenado, um bom exemplo pode ser visto na **Figura 12**.

```
public record AuthenticationDTO(
    @NotBlank @Email String email,
    @NotBlank String senha,
    @NotNull TipoPerfil perfilDesejado

) {}
```

Figura 12 – Utilização de Records do Java 21 para imutabilidade de dados em DTOs. Fonte: Autor (2026).

- **Records (Java 21):** Utilizou-se o recurso de **`Records`** do Java 21 para a criação de **DTOs**. Por serem imutáveis por padrão, eles garantem que os dados da requisição não sejam alterados acidentalmente durante o processamento.
- **Mappers:** Foram implementadas classes de mapeamento manuais ou via componentes de mapeamento, garantindo que informações sensíveis (como senhas) nunca saiam da camada de domínio para a resposta JSON da API.

4.8 Integração com MinIO e Armazenamento de Documentos

Para gerir arquivos binários de forma escalável, o sistema integra-se ao MinIO. Na **Figura 13** demonstramos o MinIO funcionando.

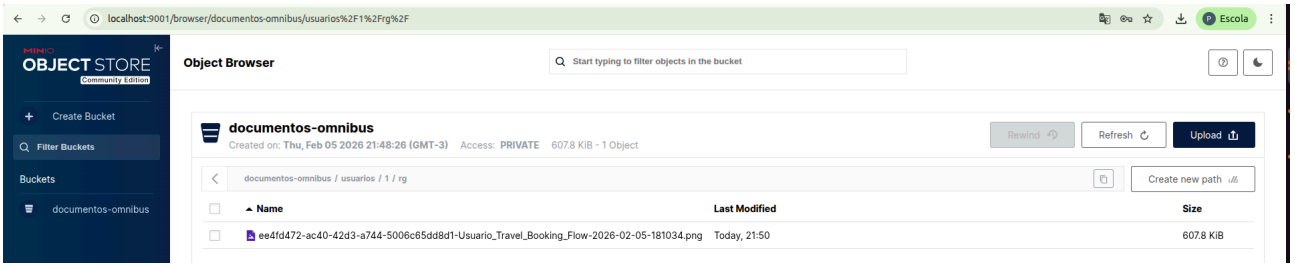


Figura 13 – Organização interna com persistência de arquivos do MinIO. Fonte: Autor (2026).

- **Configuração:** A classe **`MinioConfig`** utiliza o padrão *Bean* do Spring para disponibilizar o cliente de conexão.
- **Serviço de Documentos:** O **`DocumentoService`** gerencia o upload e a recuperação de arquivos, armazenando apenas a referência do objeto no PostgreSQL e o conteúdo binário no MinIO, mantendo o banco de dados leve.

A **Figura 14** apresenta o fluxo de upload de documentos: o usuário envia o arquivo e o tipo de documento diretamente para a API, que então realiza o envio do arquivo ao MinIO e, em seguida, salva os metadados no banco de dados. Após a confirmação dessas etapas, a API retorna ao usuário uma resposta de sucesso contendo as informações do documento armazenado.

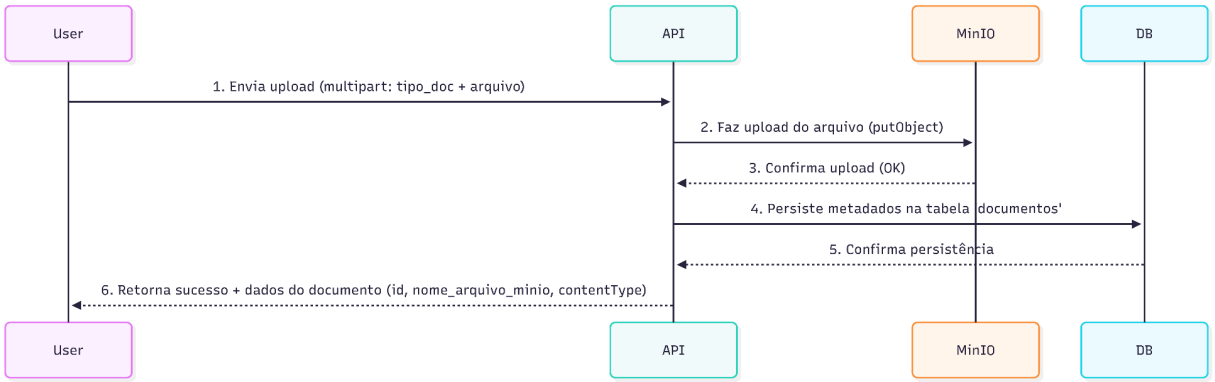


Figura 14 – Diagrama de sequência demonstrativo de upload de arquivo. Fonte: Autor(2026).

4.9 Auditoria, Logs e Observabilidade

Para garantir que o protótipo seja auditável e resiliente, mesmo em sua fase inicial de desenvolvimento, foram implementadas camadas de rastreabilidade que preparam o sistema para uma futura escala de produção.

4.9.1 Auditoria Automática com Hibernate Envers

Através do **Hibernate Envers** e da anotação **`@Auditable`**, o sistema já está preparado para monitorar a criação e alteração de usuários e empresas.

- **Rastreabilidade:** Cada inserção ou modificação gera uma revisão automática. No estágio atual do protótipo, isso permite auditar o cadastro de novas transportadoras e a atribuição de perfis de acesso, garantindo que o histórico de permissões seja preservado, assim com vemos na **Figura 15**.

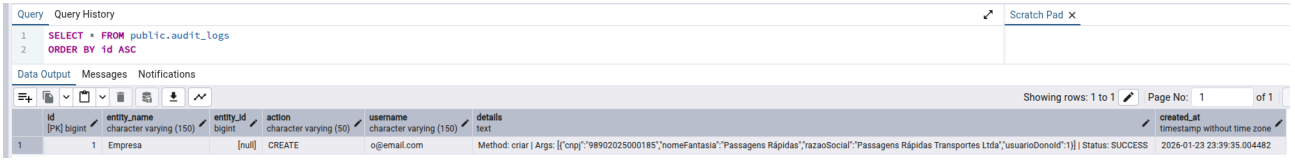


Figura 15 – Estrutura de tabelas de histórico e auditoria geradas pelo Hibernate Envers.

A **figura 16** demonstra o fluxo de auditoria com Hibernate Envers, que inicia quando o controller recebe a requisição e delega ao service as regras de negócio. O service invoca o repositório, que aciona o Hibernate para persistir, atualizar ou remover a entidade dentro de uma sessão. Nesse momento, o listener do Envers intercepta a operação e gera registros nas tabelas de auditoria (por exemplo, *_AUD), além de criar/atualizar a entidade de revisão (RevisionEntity), que armazena metadados como usuário, data/hora e tipo de operação (ADD, MOD, DEL). Ao final, a transação é confirmada e os logs de auditoria ficam disponíveis para consultas históricas via AuditReader.

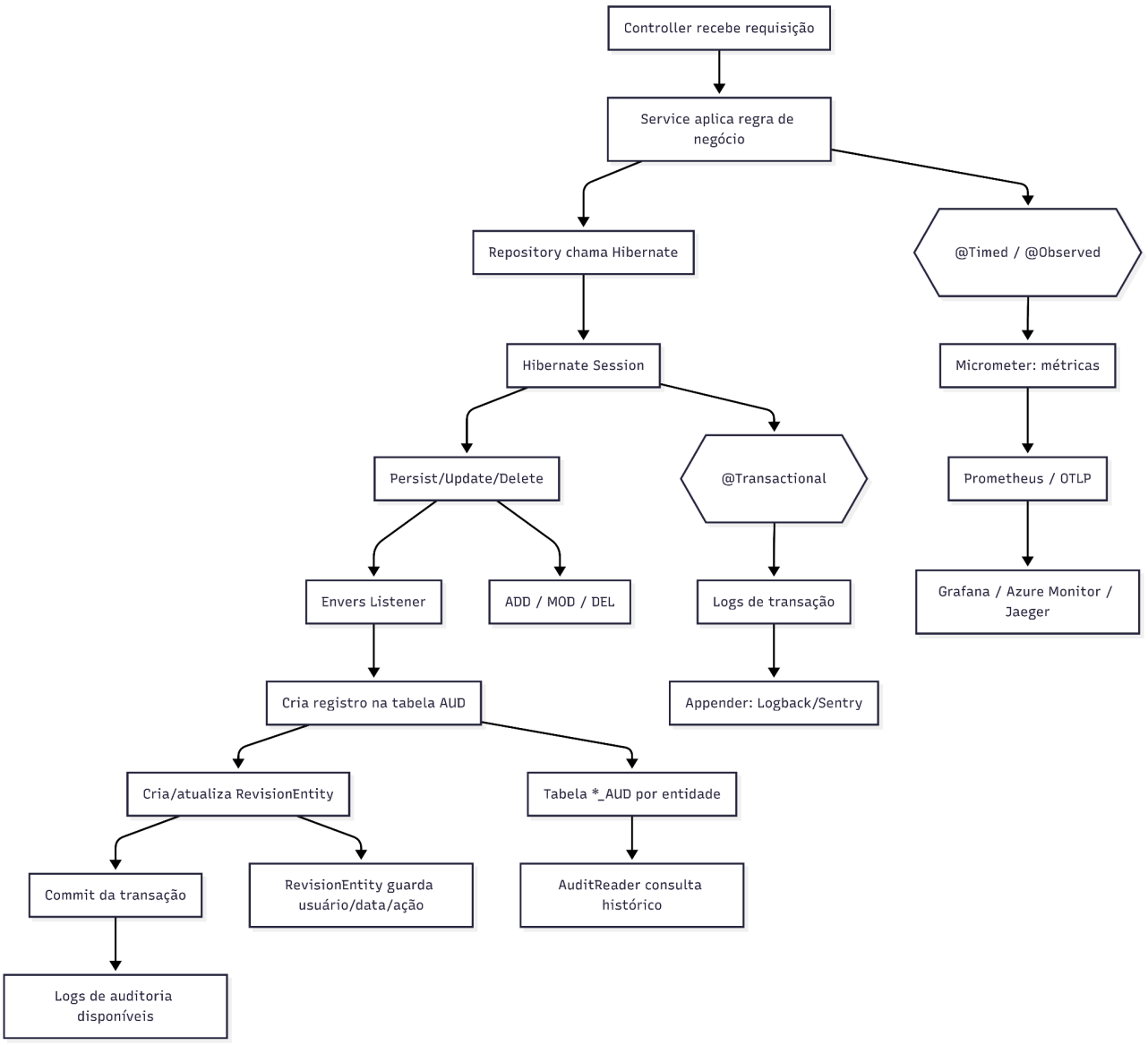


Figura 16 – Diagrama demonstrando fluxo do hibernate dentro do projeto.

Paralelamente, o monitoramento é realizado no nível do service, geralmente com anotações como @Timed ou @Observed, que publicam métricas via Micrometer para backends como Prometheus ou OTLP, permitindo visualização em ferramentas como Grafana, Azure Monitor ou Jaeger. Além disso, a camada transacional (por exemplo, @Transactional) gera logs de transação que podem ser encaminhados por appenders (Logback/Sentry), complementando a observabilidade da aplicação com dados de performance, rastreamento e auditoria.

4.9.2 Observabilidade e Diagnóstico com Sentry

Como o sistema encontra-se em fase de prototipagem, a observabilidade é vital para identificar falhas de implementação rapidamente. O **Sentry** foi configurado para capturar exceções em tempo real, permitindo que o desenvolvedor visualize o *stack trace* exato do erro ocorrido durante o fluxo de login ou cadastro, agilizando o ciclo de depuração. Um bom exemplo do que o **Sentry** monitora pode ser visto na **Figura 17.**

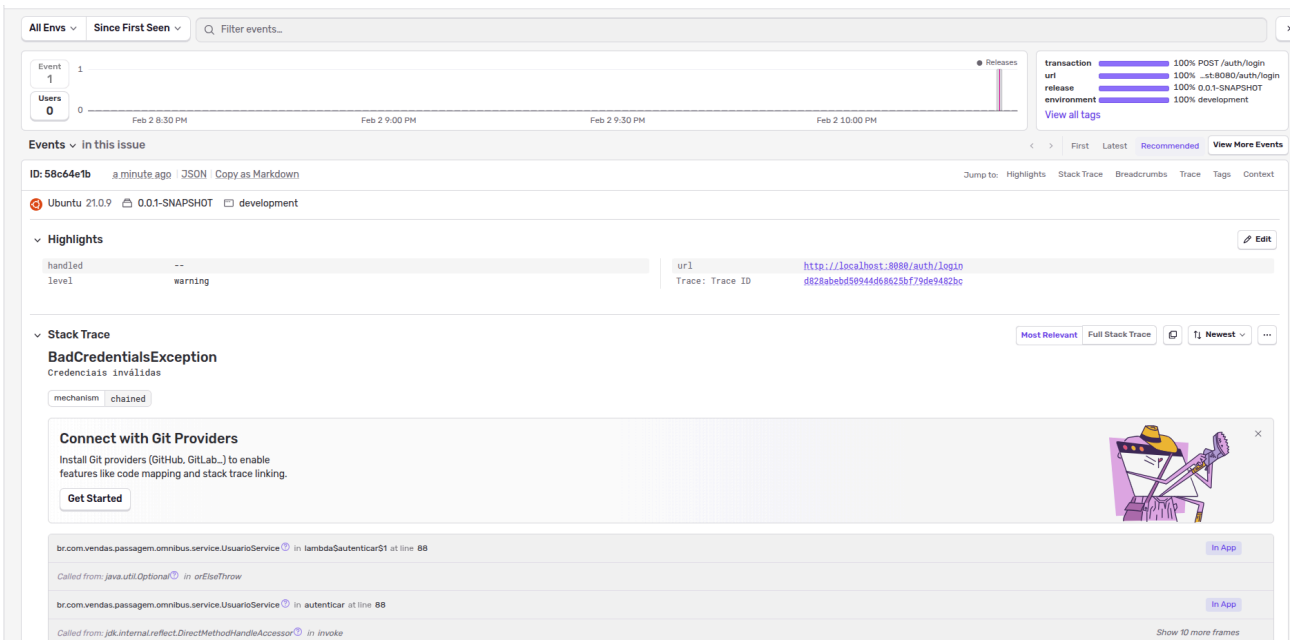


Figura 17 – Captura de exceções e diagnóstico de falhas em tempo real via Sentry.

4.10 Considerações sobre o Estágio de Desenvolvimento do Protótipo

É importante salientar que o OmniBus encontra-se em estágio de **MVP (Minimum Viable Product)** técnico. Nesta fase, o foco do desenvolvimento concentrou-se na estabilização da arquitetura, na segurança via JWT e na lógica de transição de perfis.

As funcionalidades finalizadas e testadas incluem:

- Autenticação e Autorização.
- Gestão de Identidade com seleção de perfil no login.
- Cadastro de usuários e vínculo com empresas.
- Integração com MinIO para upload de documentos de validação.

Funcionalidades como a criação de viagens (módulo empresa) e o agendamento de assentos (módulo passageiro) estão mapeadas no *backlog* do projeto como evoluções naturais, uma vez que a base arquitetural (camadas de Service e Domain) já foi preparada para suportar estas regras de negócio sem a necessidade de refatoração estrutural.

5. CONCLUSÃO

Embora o protótipo atual foque primordialmente na infraestrutura e na gestão de identidade — resolvendo o problema de rigidez de perfis observado no projeto GEPAG — a arquitetura baseada em *Clean Architecture* e Java 21 mostrou-se eficaz. O trabalho atingiu seu objetivo ao entregar uma fundação técnica sólida onde novas funcionalidades, como a reserva de passagens e a gestão de rotas, podem ser acopladas de forma plugável e segura.

Os resultados alcançados demonstram um ganho significativo de produtividade técnica. Enquanto no projeto GEPAG o estabelecimento de padrões e configurações iniciais demandou um tempo considerável e múltiplas refatorações por parte da equipe, o protótipo **OmniBus** foi estruturado em aproximadamente uma semana de tempo hábil de produção. Mesmo sendo um trabalho individual e em tempo reduzido, a arquitetura garantiu funcionalidades iniciais sólidas e um fluxo de desenvolvimento definido. Isso comprova que, ao adotar padrões claros desde o início, a curva de aprendizado sobre "onde encontrar cada componente" diminui, tornando a manutenção e a evolução do software mais rápidas e menos suscetíveis aos erros que demandaram atenção excessiva no projeto anterior.

Em suma, a disciplina na definição de padrões revelou-se o principal motor para a sustentabilidade de software acadêmico e profissional, permitindo que uma estrutura complexa seja erguida com maior agilidade e menor dívida técnica.

5.1 Trabalhos Futuros

Como continuidade deste trabalho e visando a evolução do ecossistema OmniBus, propõem-se as seguintes frentes de implementação:

- **Aprimoramento de Métricas:** Integrar o **JaCoCo** ao pipeline de integração contínua para obter métricas detalhadas de cobertura de testes, complementando a análise estática já realizada;
- **Documentação e Regras de Negócio:** Formalizar a documentação das regras de negócio específicas por serviço, garantindo que o conhecimento técnico seja facilmente transferível;
- **Preparação para Produção (Deploy):** Otimizar o reconhecimento de variáveis de ambiente para facilitar o processo de *deploy* via Docker em ambientes de produção;

- **Desenvolvimento de Front-end:** Criar uma interface de usuário (SPA ou Mobile) para consumir a API, validando a eficiência dos contratos de dados estabelecidos via Swagger;
- **Finalização de Funcionalidades:** Concluir os módulos de reserva e gestão de frotas, utilizando a base de Identidade Dinâmica já consolidada neste protótipo.

REFERÊNCIAS

BECK, Kent. Test Driven Development: by example. Boston: Addison-Wesley, 2003.

DOCKER. Docker documentation. Disponível em: https://docs.docker.com/. Acesso em: 03 fev. 2026.

FOWLER, Martin. Refatoração: aperfeiçoando o design de código existente. 1. ed. São Paulo: Addison-Wesley, 1999.

HIBERNATE. Hibernate Envers User Guide. Disponível em: https://hibernate.org/orm/envers/. Acesso em: 03 fev. 2026.

KING, Gavin; BAUER, Christian. Java Persistence with Hibernate. 2. ed. New York: Manning Publications, 2015.

MARTIN, Robert C. Clean Architecture: a craftsman's guide to software structure and design. Upper Saddle River, NJ: Prentice Hall, 2017.

MARTIN, Robert C. Código limpo: habilidades práticas do software ágil. Rio de Janeiro: Alta Books, 2009.

MINIO. MinIO Object Storage Documentation. Disponível em: https://min.io/docs/minio/linux/index.html. Acesso em: 03 fev. 2026.

ORACLE. Java SE 21 Documentation. Disponível em: https://docs.oracle.com/en/java/javase/21/. Acesso em: 03 fev. 2026.

SENTRY. Sentry Documentation. Disponível em: https://docs.sentry.io/. Acesso em: 03 fev. 2026.

SONARQUBE. SonarQube Documentation. Disponível em: https://docs.sonarqube.org/. Acesso em: 03 fev. 2026.

SPRING. Spring Framework Documentation. Disponível em: https://spring.io/projects/spring-framework. Acesso em: 03 fev. 2026.

WALLS, Craig. Spring em Ação. 5. ed. Rio de Janeiro: Alta Books, 2019.

MARTIN, Robert C. Arquitetura Limpa: O guia do artesão para estrutura e design de software. 1. ed. Rio de Janeiro: Alta Books, 2019.