INSTITUTO FEDERAL GOIANO - CAMPUS CERES

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GUSTAVO BRAYAN SILVA DA COSTA REZENDE

RENOVAR: Desenvolvimento de uma Application Programming Interface (API) REST com Kotlin e Spring Boot para gestão de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs) em obras

CERES - GO

2025

GUSTAVO BRAYAN SILVA DA COSTA REZENDE

RENOVAR: Desenvolvimento de uma Application Programming Interface (API) REST com Kotlin e Spring Boot para gestão de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs) em obras

Trabalho de conclusão de curso apresentado ao curso de Sistemas de Informação do Instituto Federal Goiano - Campus Ceres, como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação, sob orientação do Prof. Especialista Paulo Henrique Rodrigues Araujo e Co-orientação do Prof. Especialista Isaac Mendes de Melo.

CERES - GO

2025

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema Integrado de Bibliotecas do IF Goiano - SIBi

Rezende, Gustavo
R467r RENOVAR: Desenvolvimento de uma Application Programming Interface (API) REST com Kotlin e Spring Boot para gestão de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs) em obras / Gustavo Rezende. Ceres 2025.

1f. il.

Orientador: Prof. Esp. Paulo Henrique Rodrigues Araujo.
Coorientador: Prof. Esp. Isaac Mendes de Melo.
Tcc (Especialista) - Instituto Federal Goiano, curso de 0320203 - Bacharelado em Sistemas de Informação - Ceres (Campus

I. Título.

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO

PARA DISPONIBILIZAR PRODUÇÕES TÉCNICO-CIENTÍFICAS NO REPOSITÓRIO INSTITUCIONAL DO IF GOIANO

Com base no disposto na Lei Federal nº 9.610, de 19 de fevereiro de 1998, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia Goiano a disponibilizar gratuitamente o documento em formato digital no Repositório Institucional do IF Goiano (RIIF Goiano), sem ressarcimento de direitos autorais, conforme permissão assinada abaixo, para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IF Goiano.

IDENTIFICAÇÃO DA PRODUÇÃO TÉCNICO-CIENTÍFICA

- ☐ Tese (doutorado)
- ☐ Dissertação (mestrado)
- ☐ Monografia (especialização)
- ☑ TCC (graduação)
- ☐ Artigo científico
- ☐ Capítulo de livro
- ☐ Livro
- ☐ Trabalho apresentado em evento
- ☐ Produto técnico e educacional - Tipo:

Nome completo do autor: Gustavo Brayan Silva da Costa Rezende

Matrícula: 2020103202030020

Título do trabalho: RENOVAR: Desenvolvimento de uma Application Programming Interface (API) REST com Kotlin e Spring Boot para gestão de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs) em obras

RESTRIÇÕES DE ACESSO AO DOCUMENTO

Documento confidencial: ☑ Não ☐ Sim, justifique:

Informe a data que poderá ser disponibilizado no RIIF Goiano: 17 / 06 / 2025

O documento está sujeito a registro de patente? ☐ Sim ☑ Não

O documento pode vir a ser publicado como livro? ☐ Sim ☑ Não

DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O(a) referido(a) autor(a) declara:

• Que o documento é seu trabalho original, detém os direitos autorais da produção técnico-científica e não infringe os direitos de qualquer outra pessoa ou entidade;

• Que obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autoria, para conceder ao Instituto Federal de Educação, Ciência e Tecnologia Goiano os direitos requeridos e que este material cujos direitos autorais são de terceiros, estão claramente identificados e reconhecidos no texto ou conteúdo do documento entregue;

• Que cumpriu quaisquer obrigações exigidas por contrato ou acordo, caso o documento entregue seja baseado em trabalho financiado ou apoiado por outra instituição que não o Instituto Federal de Educação, Ciência e Tecnologia Goiano.

Rubiataba (Local) — 16 / 06 / 2025 (Data)

_____
Assinatura do autor e/ou detentor dos direitos autorais

Ciente e de acordo:

_____
Assinatura do(a) orientador(a)

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA GOIANO

ATA DE DEFESA DE TRABALHO DE CURSO

Ao(s) 03 dia(s) do mês de junho do ano de dois mil e vinte e cinco, realizou-se a defesa de Trabalho de Curso do(a) acadêmico(a) Gustavo Brayan Silva da Costa Rezende, do Curso de Bacharelado em Sistemas de Informação, matrícula 2020103202030020, cujo título é "RENOVAR: Desenvolvimento de uma API REST com Kotlin e Spring Boot para gestão de funcionários, ferramentas e EPIs em obras". A defesa iniciou-se às 19 horas e 43 minutos, finalizando-se às 21 horas e 48 minutos. A banca examinadora considerou o trabalho APROVADO com média 7,4 no trabalho escrito, média 7,8 no trabalho oral, apresentando assim média aritmética final de 7,6 pontos, estando o(a) estudante APTO para fins de conclusão do Trabalho de Curso.

Após atender às considerações da banca e respeitando o prazo disposto em calendário acadêmico, o(a) estudante deverá fazer a submissão da versão corrigida em formato digital (.pdf) no Repositório Institucional do IF Goiano – RIIF, acompanhado do Termo Ciência e Autorização Eletrônico (TCAE), devidamente assinado pelo autor e orientador.

Os integrantes da banca examinadora assinam a presente.

(Assinado Eletronicamente)
Paulo Henrique Rodrigues Araujo
Professor Orietador

(Assinado Eletronicamente)
Jonatas Teixeira Machado
Professor Avaliador 01

(Assinado Eletronicamente)
Roitier Campos Gonçalves
Professor Avaliador 02

Documento assinado eletronicamente por:

- **Paulo Henrique Rodrigues Araujo**, **PROF ENS BAS TEC TECNOLOGICO-SUBSTITUTO**, em 03/06/2025 22:11:06.
- **Roitier Campos Goncalves**, **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 03/06/2025 22:12:25.
- **Jonatas Teixeira Machado**, **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 03/06/2025 22:12:52.

Este documento foi emitido pelo SUAP em 03/06/2025. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse https://suap.ifgoiano.edu.br/autenticar-documento/ e forneça os dados abaixo:

Código Verificador: 713441
Código de Autenticação: ba26d825a3

RESUMO

Este trabalho documenta o desenvolvimento de uma Application Programming Interface (API) para o gerenciamento de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs) no setor de construção. O objetivo é oferecer uma solução eficiente para o controle desses recursos, garantindo maior organização, rastreabilidade e segurança no ambiente de trabalho. A Application Programming Interface (API) foi desenvolvida seguindo boas práticas de arquitetura em camadas, priorizando escalabilidade, manutenção e integração com outros sistemas. A aplicação permite o cadastro, monitoramento e controle do uso de ferramentas e EPIs, facilitando a gestão operacional e contribuindo para a otimização dos processos na construção.

Palavras-chave: Gestão de obras, Sistema de informação, Ferramentas, Equipamentos de Proteção Individual, Aplicativo de gestão

ABSTRACT

This work documents the development of an Application Programming Interface (API) for managing employees, tools, and Personal Protective Equipment (PPE) in the construction sector. The objective is to provide an efficient solution for controlling these resources, ensuring greater organization, traceability, and safety in the work environment. The Application Programming Interface (API) was developed following best practices in layered architecture, prioritizing scalability, maintainability, and integration with other systems. The application enables the registration, monitoring, and control of tool and PPE usage, facilitating operational management and contributing to the optimization of processes in construction.

Keywords: Construction management, Information system, Tools, Personal Protective Equipment, Management application

Sumário

Lista de Figuras

Lista de Tabelas

1 INTRODUÇÃO

A gestão integrada de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs) é fundamental para garantir a eficiência operacional e a segurança nas atividades da construção civil. De acordo com Trevisan (2015), a gestão de segurança no ambiente de obras envolve um conjunto de ações que incluem a análise de riscos e a definição de medidas preventivas e protetivas adequadas.

A construção civil apresenta altos índices de acidentes, sendo responsável por 5,46% dos casos registrados no Brasil, conforme dados da Associação Nacional de Medicina do Trabalho (2019). Entre os principais problemas identificados estão as quedas de altura, que correspondem a cerca de 40% dos acidentes, de acordo com informações do Sintricomb (2019), o uso inadequado de ferramentas e a falta de registros relacionados ao fornecimento e controle de Equipamentos de Proteção Individual (EPIs), como apontado por Af Figueiredo (2018).

Portanto, a contextualização do problema destaca a necessidade de soluções tecnológicas que facilitem a gestão integrada desses componentes, visando aprimorar os processos na construção e garantir a segurança e bem-estar dos trabalhadores.

2 OBJETIVOS E JUSTIFICATIVAS

2.1 OBJETIVO GERAL

Desenvolver uma API (Application Programming Interface) para o gerenciamento integrado de:

- Funcionários
- Ferramentas
- Equipamentos de Proteção Individual (EPIs)

2.2 OBJETIVOS ESPECÍFICOS

1. Facilitar a gestão de funcionários em obras
2. Monitorar estado e disponibilidade de equipamentos
3. Implementar integração via API REST

2.3 JUSTIFICATIVAS

O setor da construção enfrenta desafios significativos no controle integrado de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs). A motivação para o desenvolvimento deste trabalho surgiu da observação direta, por parte

do pesquisador, de falhas recorrentes nesse controle no ambiente de trabalho em que está inserido. A ausência de um sistema eficiente para o gerenciamento desses recursos tem gerado problemas como extravio de ferramentas, distribuição inadequada de EPIs e dificuldade na alocação de funcionários, afetando tanto a produtividade quanto a segurança no canteiro de obras.

Diante desse contexto, este projeto visa contribuir com uma solução prática e aplicável à realidade do setor, promovendo melhorias na organização dos processos e no cumprimento das normas de segurança. A relevância social da pesquisa reside na possibilidade de reduzir acidentes de trabalho, aumentar a eficiência operacional e facilitar a rastreabilidade de recursos humanos e materiais, o que pode beneficiar empresas do setor e, indiretamente, os trabalhadores.

Para atender a essas demandas, o desenvolvimento de uma ferramenta API RESTful utilizando Kotlin e Spring Boot surge como uma solução tecnológica estratégica. A escolha da arquitetura de software deve ser orientada pelos requisitos de qualidade do sistema, especialmente em domínios complexos como o da construção, conforme discutido por Sommerville (2011).

A adoção da arquitetura em camadas neste projeto segue o princípio de separação de interesses, que promove a modularização do sistema e facilita a manutenção e a evolução do software. Essa abordagem se concretiza na divisão entre a camada de apresentação (controllers REST), a camada de serviço (lógica de negócio) e a camada de persistência (acesso a dados), alinhando-se às práticas recomendadas por Sommerville (2011).

A escolha do Kotlin como linguagem de desenvolvimento alinha-se com a ênfase de Sommerville (2011) na importância de utilizar linguagens que promovam segurança de tipos e expressividade. Recursos como null-safety, interoperabilidade com Java e suporte a corrotinas para programação assíncrona tornam o Kotlin particularmente adequado para o desenvolvimento de APIs robustas.

O Spring Boot foi selecionado por encapsular padrões arquiteturais comprovados, reduzindo riscos técnicos ao oferecer soluções pré-validadas para problemas comuns. Seus benefícios incluem injeção de dependências automatizada, configuração simplificada e integração nativa com sistemas de persistência, características valorizadas por Sommerville (2011).

Para o armazenamento de dados, o PostgreSQL foi escolhido por seu rigoroso suporte a transações ACID, característica essencial para sistemas de gestão crítica como o proposto. A consistência dos dados é um requisito fundamental em sistemas que gerenciam recursos físicos e humanos, conforme destacado por Sommerville (2011).

Essa combinação tecnológica —Kotlin, Spring Boot e PostgreSQL— forma uma base sólida para a API, permitindo não apenas o gerenciamento eficiente de funcionários, ferramentas e EPIs, mas também a evolução futura do sistema. A arquitetura deve antecipar necessidades futuras sem sacrificar a simplicidade atual, princípio enfatizado por Sommerville (2011) que guiou todas as decisões técnicas deste projeto.

3 METODOLOGIA

Optou-se pela pesquisa aplicada. A pesquisa aplicada foi escolhida por buscar solucionar um problema no setor da construção.

Os requisitos do sistema foram levantados de forma empírica, com base na experiência do pesquisador na área de atuação, por meio de observações diretas e questionamentos informais com colegas de trabalho. O levantamento baseou-se na vivência prática das rotinas e necessidades enfrentadas no setor de obras mecânicas, permitindo identificar funcionalidades essenciais ao sistema proposto.

Quanto ao fluxo de desenvolvimento, o processo ocorreu de maneira linear e contínua, com as funcionalidades sendo desenvolvidas à medida que iam sendo idealizadas, percebidas como necessárias e discutidas com colegas de trabalho. Dessa forma, pode-se dizer que o desenvolvimento seguiu uma abordagem intuitiva e discutida, com foco na solução de problemas reais observados no ambiente de trabalho.

3.1 DEFINIÇÕES DOS REQUISITOS DA APLICAÇÃO

3.1.1 Requisitos Funcionais

Tabela 1: Requisitos funcionais do sistema

Código	**Nome**	**Descrição**
RF-01	Criar endereço	O sistema deve criar um novo endereço
RF-02	Atualizar endereço	O sistema deve atualizar um endereço existente

Código	Nome	Descrição
RF-03	Buscar endereço por ID	O sistema deve buscar um endereço por ID
RF-04	Criar função	O sistema deve criar uma nova função
RF-05	Buscar função por ID	O sistema deve buscar uma função por ID
RF-06	Buscar função por nome	O sistema deve buscar uma função por nome
RF-07	Listar funções	O sistema deve listar todas as funções
RF-08	Criar funcionário	O sistema deve criar um novo funcionário
RF-09	Listar funcionários	O sistema deve listar todos os funcionários
RF-10	Buscar funcionário por ID	O sistema deve buscar um funcionário por ID
RF-11	Atualizar funcionário	O sistema deve atualizar um funcionário
RF-12	Buscar funcionário por status	O sistema deve buscar funcionários por status
RF-13	Buscar funcionário por nome	O sistema deve buscar funcionários por nome
RF-14	Adicionar EPI a funcionário	O sistema deve adicionar um EPI a um funcionário
RF-15	Devolver EPI de funcionário	O sistema deve devolver um EPI de um funcionário
RF-16	Listar EPIs	O sistema deve listar todos os EPIs
RF-17	Buscar EPI de funcionário por ID	O sistema deve buscar um EPI de um funcionário por ID
RF-18	Listar EPIs entregues	O sistema deve listar todos os EPIs com status ”entregue”
RF-19	Listar EPIs devolvidos	O sistema deve listar todos os EPIs com status ”devolvido”
RF-20	Devolver todos os EPIs	O sistema deve devolver todos os EPIs de um funcionário
RF-21	Listar EPIs por funcionário	O sistema deve listar todos os EPIs de um funcionário

Código	Nome	Descrição
RF-22	Funcionários por EPI específico	O sistema deve listar todos os funcionários que possuem um EPI específico
RF-23	EPIs entregues por funcionário	O sistema deve listar todos os EPIs entregues para um funcionário específico
RF-24	Adicionar ferramenta	O sistema deve adicionar uma ferramenta a um funcionário
RF-25	Devolver ferramenta	O sistema deve devolver uma ferramenta de um funcionário
RF-26	Listar ferramentas emprestadas	O sistema deve listar todas as ferramentas emprestadas
RF-27	Ferramentas por funcionário	O sistema deve listar todas as ferramentas emprestadas por um funcionário
RF-28	Ferramentas emprestadas	O sistema deve listar todas as ferramentas emprestadas por um funcionário com status ”emprestado”
RF-29	Ferramentas devolvidas	O sistema deve listar todas as ferramentas emprestadas por um funcionário com status ”devolvido”
RF-30	Adicionar funcionário à obra	O sistema deve adicionar um funcionário a uma obra
RF-31	Devolver funcionário da obra	O sistema deve devolver um funcionário de uma obra
RF-32	Listar funcionários em obras	O sistema deve listar todos os funcionários em obras
RF-33	Buscar funcionário em obra	O sistema deve buscar um funcionário em uma obra por ID
RF-34	Funcionários trabalhando	O sistema deve listar todos os funcionários que estão trabalhando
RF-35	Funcionários não trabalhando	O sistema deve listar todos os funcionários que não estão trabalhando

Código	Nome	Descrição
RF-36	Obra atual do funcionário	O sistema deve listar a obra em que um funcionário está trabalhando
RF-37	Funcionários em obra específica	O sistema deve listar todos os funcionários que estão trabalhando em uma obra específica
RF-38	Salvar novo EPI	O sistema deve salvar um novo EPI
RF-39	Atualizar EPI	O sistema deve atualizar um EPI existente
RF-40	Listar EPIs expirados	O sistema deve listar todos os EPIs expirados
RF-41	Listar todos os EPIs	O sistema deve listar todos os EPIs
RF-42	Buscar EPI por ID	O sistema deve buscar um EPI por ID
RF-43	Buscar EPI por nome	O sistema deve buscar um EPI por nome
RF-44	Listar EPIs não expirados	O sistema deve listar todos os EPIs não expirados
RF-45	Salvar nova ferramenta	O sistema deve salvar uma nova ferramenta
RF-46	Atualizar ferramenta	O sistema deve atualizar uma ferramenta existente
RF-47	Listar ferramentas	O sistema deve listar todas as ferramentas
RF-48	Buscar ferramenta por ID	O sistema deve buscar uma ferramenta por ID
RF-49	Buscar ferramenta por nome	O sistema deve buscar uma ferramenta por nome
RF-50	Listar ferramentas por status	O sistema deve listar ferramentas por status
RF-51	Deletar ferramenta	O sistema deve deletar uma ferramenta
RF-52	Listar ferramentas ativas	O sistema deve listar todas as ferramentas, exceto as deletadas
RF-53	Salvar ferramenta em obra	O sistema deve salvar uma nova ferramenta em uma obra

Código	Nome	Descrição
RF-54	Atualizar ferramenta em obra	O sistema deve atualizar uma ferramenta em uma obra
RF-55	Listar ferramentas por obra	O sistema deve listar todas as ferramentas de uma obra
RF-56	Ferramentas da obra por ID	O sistema deve listar todas as ferramentas de uma obra por ID
RF-57	Ferramentas emprestadas na obra	O sistema deve listar todas as ferramentas de uma obra por ID e status ”emprestado”
RF-58	Ferramentas devolvidas na obra	O sistema deve listar todas as ferramentas de uma obra por ID e status ”devolvido”
RF-59	Buscar ferramenta em obra	O sistema deve listar uma ferramenta específica em uma obra
RF-60	Criar obra	O sistema deve criar uma nova obra
RF-61	Atualizar obra	O sistema deve atualizar uma obra existente
RF-62	Listar obras	O sistema deve listar todas as obras
RF-63	Buscar obra por ID	O sistema deve buscar uma obra por ID
RF-64	Buscar obra por status	O sistema deve buscar obras por status
RF-65	Buscar obra por empresa	O sistema deve buscar obras por empresa fornecedora

Fonte: Próprio Autor.

3.1.2 Requisitos Não-Funcionais

Tabela 2: Requisitos não-funcionais do sistema

Código	Nome	Descrição
RNF-01	Guia de estilo Kotlin	O código-fonte da aplicação deve seguir o guia de estilo Kotlin da JetBrains

Código	Nome	Descrição
RNF-02	Boas práticas Spring Boot	O código deve aderir às boas práticas de codificação em Spring Boot
RNF-03	Estrutura para testes	A aplicação deve ser projetada para facilitar a escrita de testes unitários
RNF-04	Ferramentas de teste	Devem ser utilizadas as estruturas de teste JUnit para testes
RNF-05	Documentação da API	A API deve possuir documentação gerada automaticamente utilizando a biblioteca Swagger3 para Spring Boot
RNF-06	Geração de logs	Logs detalhados devem ser gerados para fins de auditoria e solução de problemas
RNF-07	Conteúdo dos logs	Os logs devem incluir informações relevantes, como: Tempo de execução, métodos chamados, erros e exceções, dados de entrada e saída (quando aplicável)
RNF-08	Controle de versão	O código-fonte da aplicação deve ser gerenciado utilizando um sistema de controle de versão, como Git
RNF-09	Boas práticas de Git Flow	O uso de branches, pull requests e boas práticas de Git Flow deve ser adotado
RNF-10	Uso de IA generativa	Ferramentas de IA generativa, como ChatGPT e GitHub Copilot, devem ser utilizadas para otimização de códigos e melhorias na produtividade durante o desenvolvimento da aplicação

Fonte: Próprio Autor.

3.2 INGLÊS COMO LINGUAGEM PADRÃO DE DESENVOLVIMENTO

Conforme SantosS e Diegues (2021), o domínio do inglês é essencial para profissionais de TI. A adoção do inglês como linguagem padrão justifica-se por:

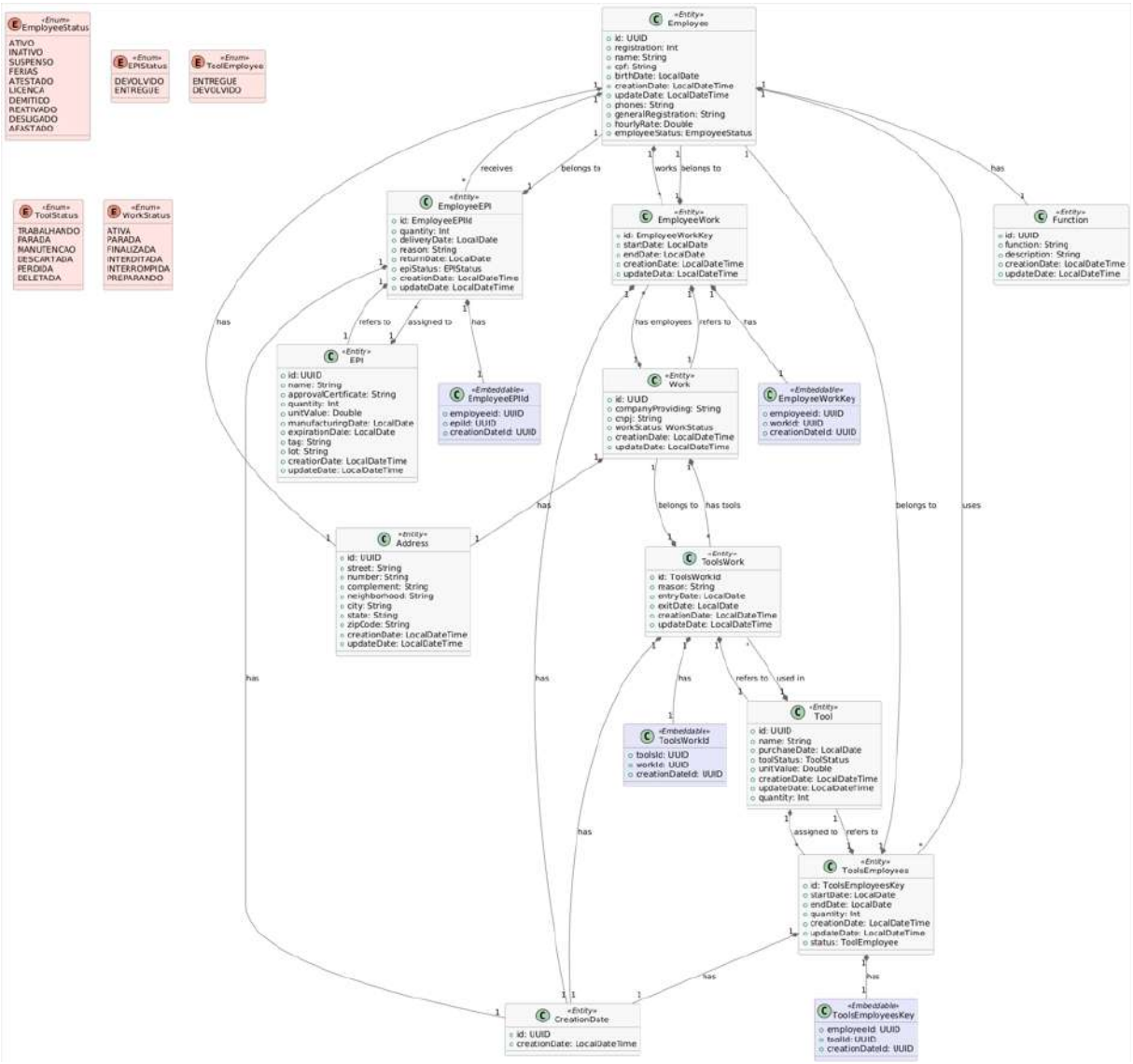
- Facilitar o acesso a documentações técnicas

- Permitir participação em comunidades globais
- Promover padronização do código
- Viabilizar internacionalização da aplicação

3.3 DIAGRAMAÇÃO UML

3.3.1 Diagrama de Classes

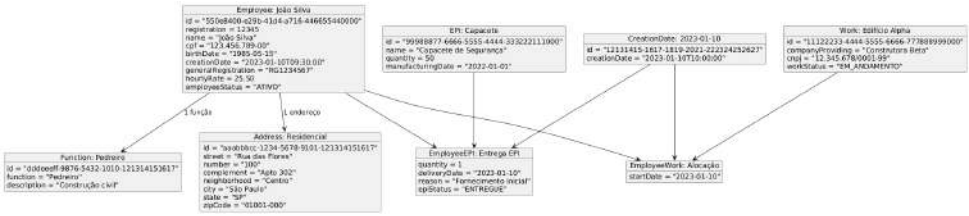
Figura 1: Diagrama de Classes



Fonte: Próprio Autor.

3.3.2 Diagrama de Objetos

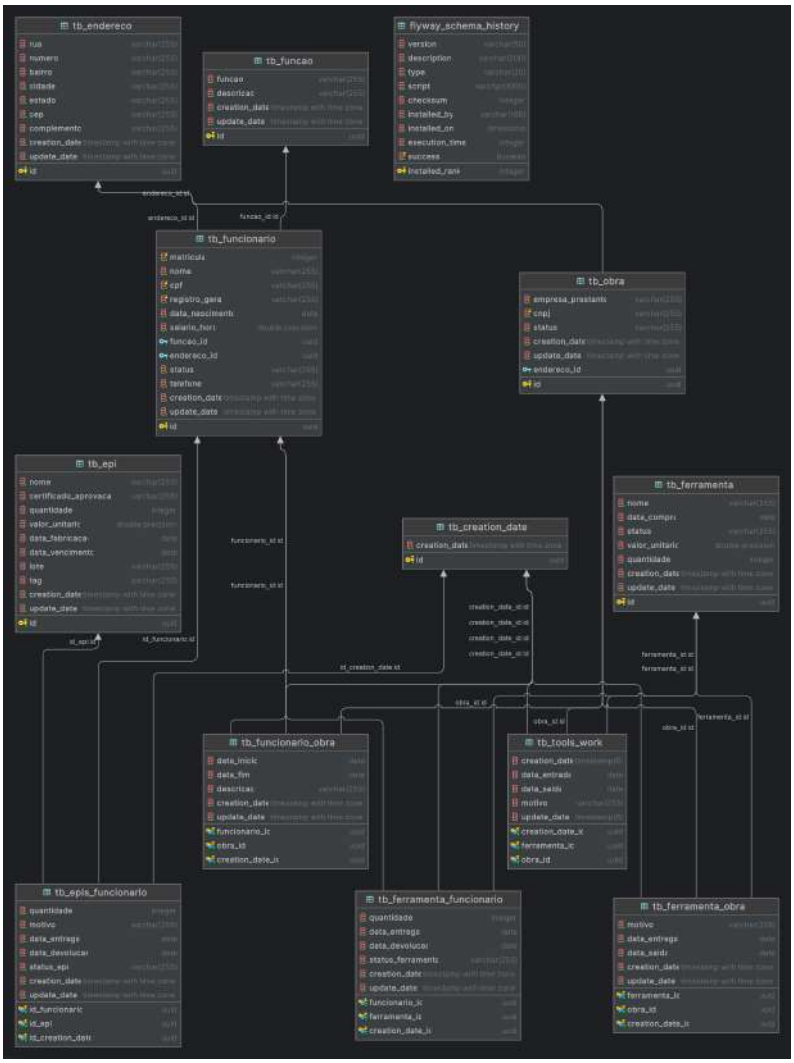
Figura 2: Diagrama de Objetos



Fonte: Próprio Autor.

3.3.3 Diagrama de Banco de Dados

Figura 3: Diagrama de Banco de Dados



Fonte: Próprio Autor.

3.4 ARQUITETURA DO SISTEMA

A arquitetura do sistema foi planejada com foco na organização modular, na manutenibilidade do código e na aplicação de boas práticas de engenharia de software. Segundo Sommerville (2011), uma arquitetura bem estruturada facilita a manutenção e evolução do sistema, promovendo a modularidade e a reutilização de componentes. Esta seção apresenta a estrutura arquitetural adotada, com ênfase na definição de pacotes e camadas que compõem a aplicação, bem como nos padrões de projeto utilizados para garantir coesão, reutilização e flexibilidade. O objetivo é demonstrar como essas decisões arquiteturais contribuem para um desenvolvimento mais eficiente e sustentável ao longo do ciclo de vida do sistema.

3.4.1 Definição de pacotes e camadas

Para definir a arquitetura da aplicação, foi escolhida a Arquitetura em Camadas, que é um estilo arquitetural que organiza o software em níveis hierárquicos, cada um com responsabilidades específicas e bem definidas. Essa separação permite que cada camada interaja apenas com suas camadas adjacentes, promovendo um acoplamento fraco entre os componentes e aumentando a coesão interna de cada módulo. Segundo Marmsoler, Malkis e Eckhardt (2015), uma arquitetura em camadas consiste em uma hierarquia de camadas, na qual os serviços se comunicam por meio de portas, e cada camada é modelada como uma relação entre serviços utilizados e fornecidos.

A adoção da arquitetura em camadas neste projeto visa garantir a separação de responsabilidades, facilitando a manutenção, escalabilidade e reutilização do código. Ao isolar as funcionalidades em diferentes camadas, é possível modificar ou expandir partes do sistema com impacto mínimo nas demais, promovendo uma estrutura mais organizada e eficiente.

3.4.2 Escolha dos Padrões de Projetos

Segundo Shvets (2021), o padrão *Bridge* é uma estrutura de design que possibilita a divisão de uma classe extensa ou de um conjunto de classes intimamente conectadas em duas hierarquias distintas - abstração e implementação - permitindo que sejam desenvolvidas de forma independente uma da outra e se comuniquem por meio de interface posteriormente.

Com a escolha da Arquitetura em Camadas para o desenvolvimento, o padrão de projeto principal adotado foi o *Bridge*. Esse padrão oferece uma solução elegante para que diferentes componentes da aplicação se comuniquem entre si por meio de interfaces, podendo até mesmo ter múltiplas implementações para uma mesma interface. Isso torna a aplicação mais flexível e facilita a implementação de outros padrões, como o *Strategy*, tornando-a mais dinâmica.

No contexto do framework Spring Boot, a implementação desse padrão é facilitada pela inversão de dependências automatizada, que é realizada pelo próprio core do framework. Isso proporciona uma integração suave e eficaz entre os componentes da aplicação, sem a necessidade de gerenciamento manual das dependências.

Para a comunicação com o banco de dados, foi utilizado o JPA (Java Persistence API) juntamente com o Spring Data JPA, que é uma solução eficaz para persistência de dados em aplicações Java. A utilização do JPA Repository simplifica a criação e execução de operações de CRUD (Create, Read, Update, Delete) em entidades, permitindo a interação com o banco de dados de forma eficiente e com menos código. A abstração fornecida pelo JPA Repository permite que o desenvolvedor foque nas regras de negócio sem se preocupar com a implementação direta de consultas SQL.

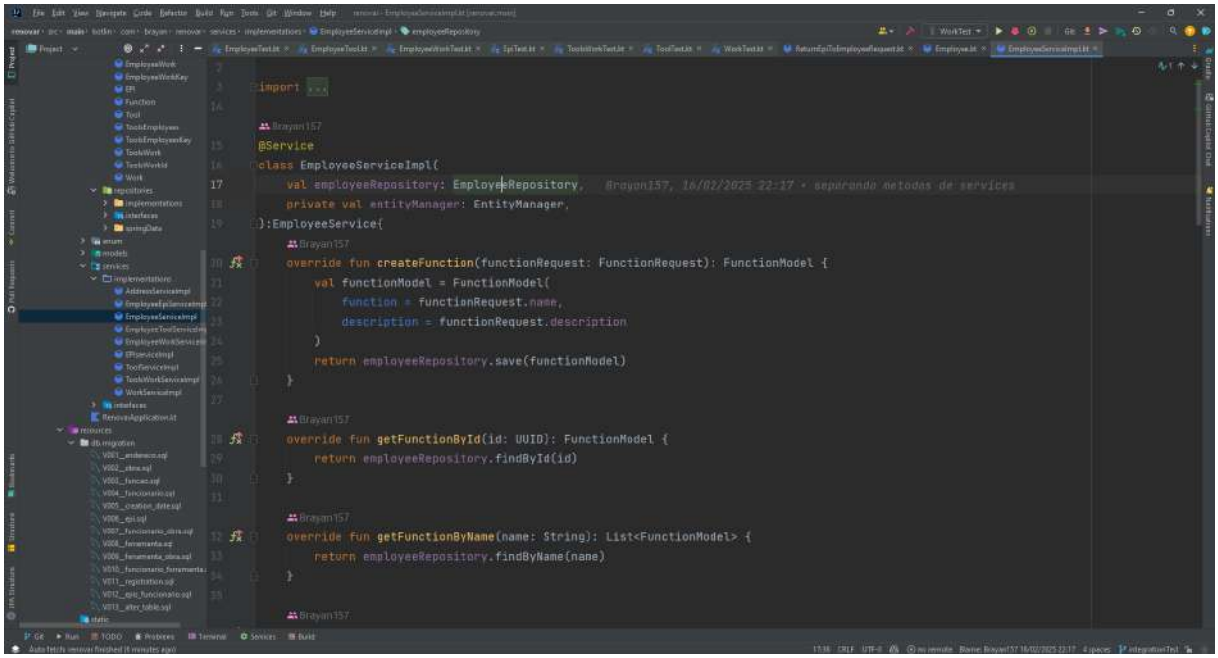
Além disso, o JPA também oferece recursos como mapeamento objeto-relacional, permitindo que as entidades Java sejam automaticamente mapeadas para tabelas no banco de dados. Isso facilita a gestão das entidades e a persistência dos dados de forma transparente.

Outro padrão de projeto utilizado na aplicação foi o *Singleton*, implementado pelo próprio framework Spring Boot. Esse padrão garante que, em toda a aplicação, haverá uma única instância de cada classe gerenciada pelo Spring. Por exemplo, quando uma classe como EmployeeServiceImpl é anotada com @Service, ela se torna um Bean gerenciado pelo Spring. Assim, o framework cria e gerencia uma única instância dessa classe, permitindo a injeção de dependências de forma automática. Esse comportamento facilita a usabilidade das classes e a criação de testes automatizados, garantindo que a mesma instância seja utilizada em todas as partes do sistema onde for necessária.

3.5 O AMBIENTE DE DESENVOLVIMENTO

Para o desenvolvimento da aplicação, foi utilizada a IDE IntelliJ IDEA Ultimate 2023.1.2, obtida por meio da licença estudantil fornecida pela JetBrains. A adoção deste ambiente proporcionou diversas vantagens. O IntelliJ IDEA oferece recursos avançados que facilitam a estruturação e organização do código, permitindo uma gestão mais ágil e intuitiva de classes e pacotes. A IDE disponibiliza ferramentas robustas para refatoração de código, simplificando a melhoria e a manutenção do software ao longo do desenvolvimento.

Figura 4: IntelliJ IDE



Fonte: Próprio Autor.

3.6 ARQUITETURA DA API

3.6.1 API REST

Uma API REST (Representational State Transfer) é uma interface que utiliza o Protocolo de Transferência de Hipertexto (HTTP), que é o protocolo padrão da web para troca de dados entre cliente e servidor. Segundo Fielding (2000), esse modelo arquitetural segue um conjunto de restrições, como a separação entre cliente e servidor, o que permite que cada parte evolua de forma independente. Além disso, a comunicação deve ser stateless, ou seja, cada requisição precisa conter todas as informações necessárias para seu processamento, sem depender de informações armazenadas no

servidor. As respostas devem ser passíveis de armazenamento em cache, permitindo seu reaproveitamento e melhorando o desempenho. A arquitetura também deve adotar uma interface uniforme, facilitando a interação entre os componentes do sistema. Outro princípio é o uso de um sistema em camadas, em que cada nível pode funcionar de maneira independente, favorecendo a escalabilidade. Por fim, de forma opcional, o servidor pode fornecer código executável ao cliente, como scripts, para ampliar suas funcionalidades.

As operações básicas de uma API REST seguem o padrão CRUD:

Tabela 3: Operações REST

Método HTTP	**Operação**	**Descrição**
POST	Create	Criação de recursos
GET	Read	Leitura de recursos
PUT	Update	Atualização de recursos
DELETE	Delete	Remoção de recursos

Fonte: Próprio Autor.

A adoção da arquitetura REST proporciona diversas vantagens ao projeto, alinhadas às restrições propostas por Fielding (2000). Entre elas, destaca-se a comunicação padronizada entre sistemas heterogêneos, viabilizada pela interface uniforme que permite a interação consistente entre diferentes tecnologias, independentemente de suas implementações internas. A escalabilidade também é favorecida pela característica stateless do REST, que possibilita a distribuição eficiente de requisições entre múltiplos servidores. Além disso, a utilização de formatos padronizados como JSON contribui para a interoperabilidade entre plataformas e linguagens distintas. Por fim, a simplicidade e a padronização do modelo REST facilitam a integração com ecossistemas modernos, tornando o desenvolvimento e a manutenção de sistemas mais ágeis e eficazes.

Conforme destacado por Fielding (2000), a utilização de uma interface uniforme e a separação entre cliente e servidor são fundamentais para a construção de sistemas distribuídos escaláveis e modulares.

3.6.2 Arquitetura em Camadas

A arquitetura em camadas adotada neste projeto fundamenta-se nos princípios de engenharia de software propostos por Sommerville (2011), particularmente em sua abordagem sobre sistemas modulares. Segundo o autor, a eficácia dessa arquitetura reside na capacidade de isolar funcionalidades em níveis distintos, permitindo que cada camada se desenvolva de maneira independente sem comprometer a integridade do sistema como um todo.

A camada de apresentação a qual foi nomeada de `controller` foi concebida seguindo os preceitos de Sommerville (2011) sobre interfaces bem definidas, atuando como único ponto de entrada para todas as requisições HTTP. Nesta camada, os controllers REST cumprem o papel de mediadores entre o mundo externo e a lógica de negócio, responsabilizando-se pela validação inicial dos dados e formatação das respostas. Essa separação rigorosa entre interface e núcleo da aplicação reflete a recomendação de Sommerville para sistemas que demandam alta manutenibilidade.

No núcleo do sistema a qual foi nomeada de `services`, a camada de serviço incorpora as observações de Sommerville (2011) sobre encapsulamento de regras de negócio. Cada serviço foi projetado como uma unidade autônoma, com interfaces claras e responsabilidades bem delimitadas, seguindo o princípio da separação de interesses que o autor considera fundamental para arquiteturas sustentáveis. A implementação de padrões como Bridge e Strategy nesta camada materializa as recomendações de Sommerville (2011) para sistemas que necessitam de flexibilidade operacional.

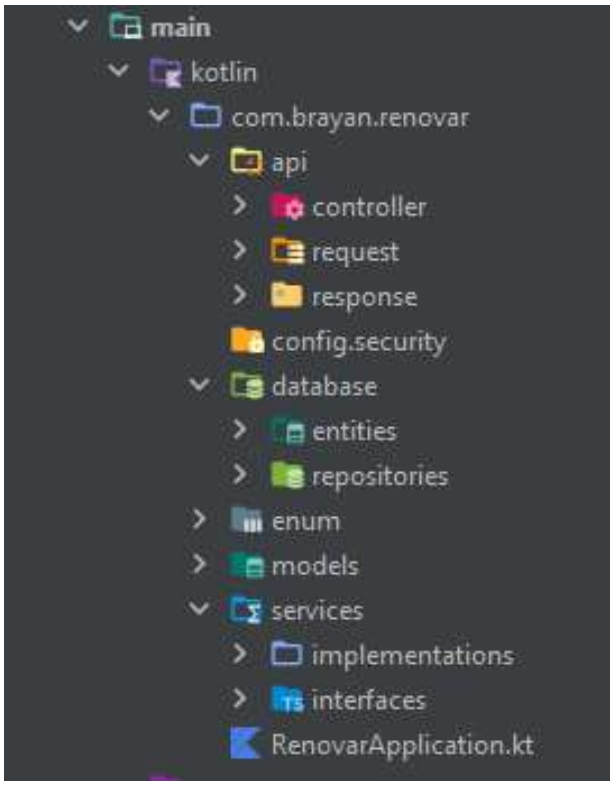
A camada de persistência a qual foi nomeada de `repositories` foi estruturada conforme os conceitos de Sommerville (2011) sobre abstração de dados, utilizando o Spring Data JPA para ocultar completamente os detalhes de implementação do banco de dados. Essa abordagem, que o autor descreve como essencial para sistemas evolutivos, permite que a aplicação mantenha consistência mesmo em cenários de mudança de tecnologias de armazenamento.

A interação entre as camadas segue o modelo de dependência unidirecional defendido por Sommerville (2011), criando um fluxo hierárquico onde: a apresentação depende apenas dos serviços, os serviços dependem da persistência e a persistência mantém-se isolada das demais.

Essa organização, ilustrada na Figura 5, resulta no que Sommerville (2011)

classifica como arquitetura em camadas estrita, particularmente adequada para sistemas com requisitos complexos de evolução. Os benefícios observados incluem: manutenção facilitada por módulos coesos, testabilidade aprimorada através de interfaces bem definidas e capacidade de substituição tecnológica por camada.

Figura 5: Estrutura de Camadas da Aplicação



Fonte: Próprio Autor.

A definição cuidadosa dos contratos entre camadas, aspecto que Sommerville destaca como crítico para o sucesso arquitetural, foi alcançada através de: interfaces Java com métodos específicos por domínio, objetos de transferência de dados imutáveis e hierarquia de exceções customizadas.

Como resultado final, a arquitetura implementada concretiza os princípios defendidos por Sommerville para sistemas de média complexidade, oferecendo a combinação ideal entre estruturação rígida e flexibilidade evolutiva necessária para o domínio da construção.

3.7 DOCKERIZAÇÃO DO BANCO DE DADOS

De acordo com a documentação oficial, o Docker é uma plataforma de containerização que permite empacotar softwares inteiros em contêineres. A principal

vantagem de utilizar essa ferramenta é a simplificação do processo de configuração do sistema, que se torna quase nulo quando as configurações estão definidas no arquivo Docker Compose. Além disso, o Docker garante a uniformidade das versões das dependências, como o banco de dados, pois, ao serem conteinerizadas, suas versões são controladas diretamente pelo arquivo de configuração do Docker, prevenindo problemas decorrentes de atualizações incompatíveis dessas dependências.

A utilização do Docker para o gerenciamento do banco de dados neste projeto foi motivada por diversas vantagens destacadas pela própria documentação oficial Docker Inc. (2023). Uma das vantagens é o isolamento do ambiente, já que o contêiner mantém o banco de dados separado do sistema hospedeiro, prevenindo conflitos de configuração. Além disso, a portabilidade é um fator importante, pois a imagem Docker pode ser executada de forma idêntica em qualquer sistema compatível. O controle de versão também é facilitado, uma vez que é possível especificar com precisão a versão do PostgreSQL utilizada no ambiente. A configuração rápida é outra vantagem relevante, permitindo a inicialização do banco de dados com apenas um comando. Por fim, o Docker garante consistência entre os ambientes de desenvolvimento, assegurando que todos os colaboradores trabalhem com a mesma configuração.

3.7.1 Implementação do Contêiner

A configuração do banco de dados via Docker foi realizada através do arquivo `docker-compose.yml`, com as seguintes especificações técnicas:

Tabela 4: Configuração do Contêiner do Banco de Dados

Parâmetro	**Valor**
Imagem Utilizada	debezium/postgres:14-alpine
Porta Exposta	5432:5432
Variáveis de Ambiente	POSTGRES_USER, POSTGRES_PASSWORD
Volume Configurado	pg_data:/var/lib/postgresql/data

Fonte: Próprio Autor.

Figura 6: Contêiner Docker do Banco de Dados PostgreSQL

Fonte: Próprio Autor.

Uma das vantagem observada nesta abordagem foi a simplificação do processo de desenvolvimento.

3.7.2 Benefícios Específicos para o Projeto

A utilização do Docker trouxe os seguintes benefícios específicos para o gerenciamento do banco de dados:

- Versionamento Controlado: A versão 14 do PostgreSQL foi fixada na imagem
- Recuperação Rápida: Em caso de problemas, o banco pode ser reiniciado em segundos
- Desenvolvimento Colaborativo: Todos os membros da equipe trabalham com a mesma configuração
- Integração Contínua: Facilita a configuração de ambientes efêmeros para testes automatizados

A configuração do Docker Compose permitiu ainda a fácil replicação do ambiente em diferentes máquinas, sendo particularmente útil durante a fase de testes e na integração com um sistema CI/CD.

3.8 TESTES

De acordo com Sommerville (2011), os testes representam uma atividade crítica no processo de garantia de qualidade de software, servindo como mecanismo de verificação e validação que complementa outras técnicas formais. O autor enfatizou que um processo de teste bem estruturado deve abranger múltiplos níveis, desde a verificação de unidades individuais até a validação do sistema como um todo, sempre alinhado com os requisitos especificados.

3.8.1 Fundamentação Teórica

A importância dos testes no ciclo de desenvolvimento de software é amplamente reconhecida na literatura técnica. Segundo Myers (2004), os testes de software possuem como objetivos principais a verificação de conformidade com requisitos funcionais e não-funcionais, a identificação precoce de defeitos durante o processo de desenvolvimento, e a validação do comportamento esperado do sistema em diferentes cenários. Essa abordagem preventiva permite detectar problemas antes que eles alcancem o ambiente de produção, reduzindo custos e aumentando a qualidade do produto final.

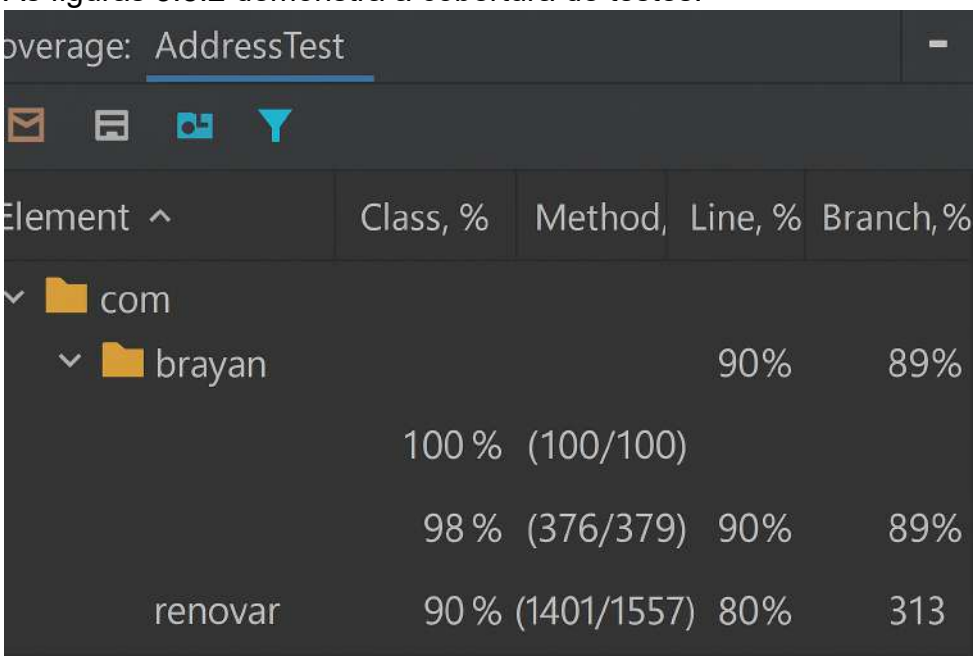
3.8.2 Abordagem de Testes

A estratégia de testes implementada neste projeto adotou uma abordagem em camadas, iniciando pelos testes unitários e evoluindo para testes de integração. Conforme as recomendações de Bitterncourt (2020), priorizamos os testes unitários como base da pirâmide de testes, estabelecendo como meta mínima a cobertura de 80% das linhas de código e 75% dos branches. Essa decisão se alinha com as práticas recomendadas por Sommerville (2011), que destaca a importância de testes unitários abrangentes para garantir a qualidade de componentes individuais antes de sua integração.

Para a implementação dos testes, utilizamos o conjunto de ferramentas de testes *spring-boot-starter-test* como base, complementado com `kotlin-test-junit5` para integração com o JUnit 5 e `mockk` para criação de mocks específicos para Kotlin. Mocks são objetos simulados usados em testes automatizados para imitar o comportamento de objetos reais, como classes, APIs ou serviços externos. Essa combinação permitiu criar testes concisos e legíveis, seguindo o princípio de Sommerville (2011) de que os testes devem ser tão cuidadosamente projetados quanto o código de produção.

Os resultados obtidos, apresentados na Tabela 5, demonstram a eficácia da abordagem adotada. Atingimos 80% de cobertura de linhas de código e 89% de cobertura de branches, com um total de 61 testes unitários implementados. Esses valores estão dentro das faixas recomendadas por Sommerville (2011) para projetos de média complexidade, garantindo um bom equilíbrio entre esforço de teste e garantia de qualidade.

As figuras 3.8.2 demonstra a cobertura de testes:



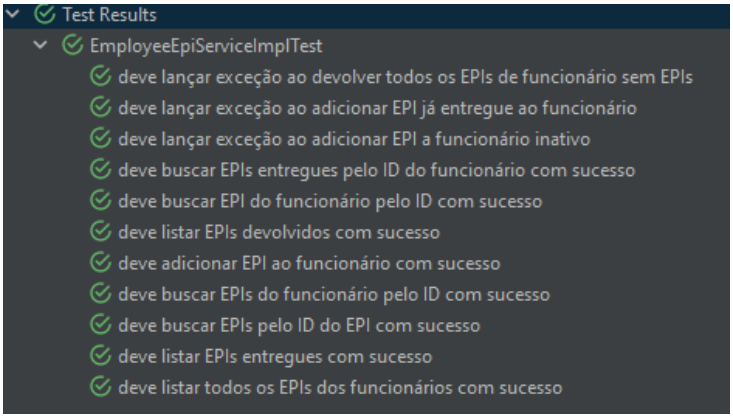
Fonte: Próprio Autor.

Tabela 5: Métricas de cobertura de testes

Métrica	**Valor**
Cobertura de Linhas	80%
Cobertura de Branches	89%
Testes Unitários	61

Fonte: Próprio Autor.

Figura 7: Testes Unitarios



Fonte: Próprio Autor.

A estratégia de testes adotada reflete as práticas consolidadas na engenharia de software, onde Sommerville (2011) argumenta que os testes devem evoluir junto com o sistema, sendo continuamente refinados à medida que novos requisitos são incorporados. Essa abordagem incremental permitiu identificar e corrigir problemas ainda nas fases iniciais de desenvolvimento, reduzindo significativamente o custo de correção de defeitos.

3.9 API

A arquitetura da API foi desenvolvida seguindo os princípios RESTful estabelecidos por Fielding (2000). O sistema trata cada entidade - EPIs, ferramentas e funcionários - como recursos únicos identificados por URIs. O endpoint `/epi` gerencia todos os equipamentos de proteção individual, enquanto `/epis/id/42` acessa um EPI específico, demonstrando a abordagem de recursos individuais.

A interface uniforme é garantida pelo uso consistente dos verbos HTTP. O método GET em `/employees` lista todos os funcionários, enquanto POST adiciona novos registros e o PUT faz alterações nos registros já existentes.

A API retorna os EPIs já associados a cada funcionário quando se acessa `employeeEpi/all` ou `employeeEpi/id`, eliminando a necessidade de navegação adicional por links relacionados. As respostas seguem um padrão JSON estruturado, com códigos de status HTTP precisos: 200 para operações bem-sucedidas, 201 para criação de recursos, 400 para requisições inválidas e 404 para recursos não encontrados, garantindo clareza na comunicação com os clientes da API.

3.9.1 Documentação Swagger da API

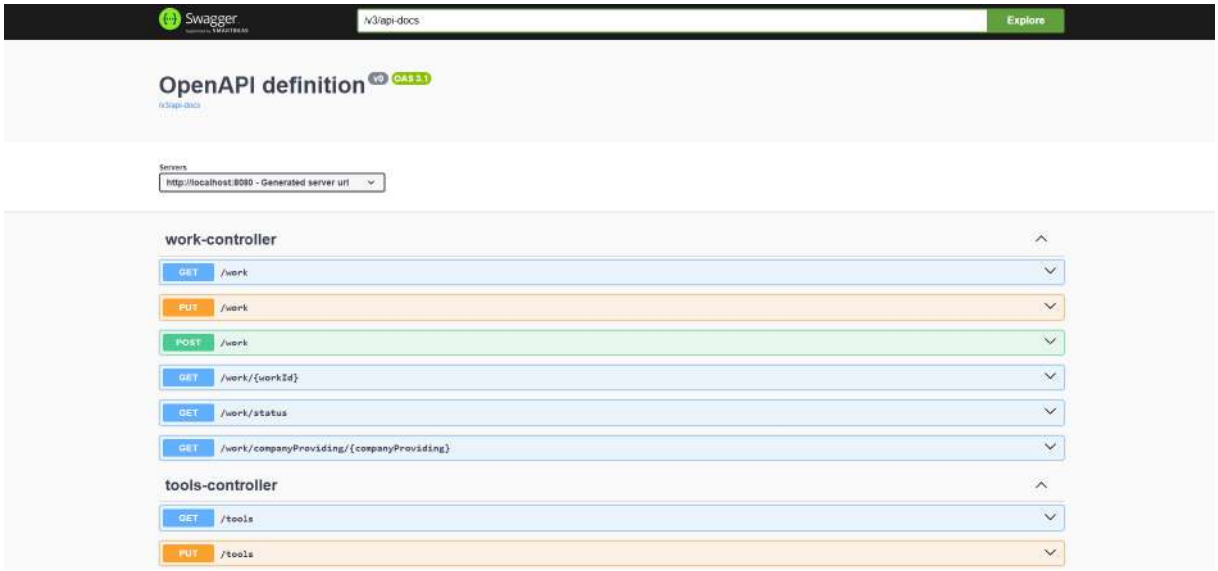
A documentação da API foi desenvolvida utilizando o framework Swagger UI, uma ferramenta amplamente adotada para documentação de APIs REST conforme padrão OpenAPI Initiative (2023). O Swagger UI gera automaticamente uma interface interativa que descreve:Todos os endpoints disponíveis na API, os parâmetros exigidos por cada operação, os modelos de dados de entrada e saída e exemplos práticos de requisições e respostas.

A Figura 8 mostra a interface principal da documentação, acessível através do endpoint `/swagger-ui/index.html`. Esta abordagem segue as recomendações de Smith e Johnson (2022) sobre documentação automatizada de APIs, que destaca três

vantagens principais: 1 - Sincronização automática: A documentação é gerada diretamente do código-fonte, evitando desatualizações, 2 - Interatividade: Permite testar os endpoints diretamente na interface e 3 - Padronização: Segue as especificações do OpenAPI 3.0.

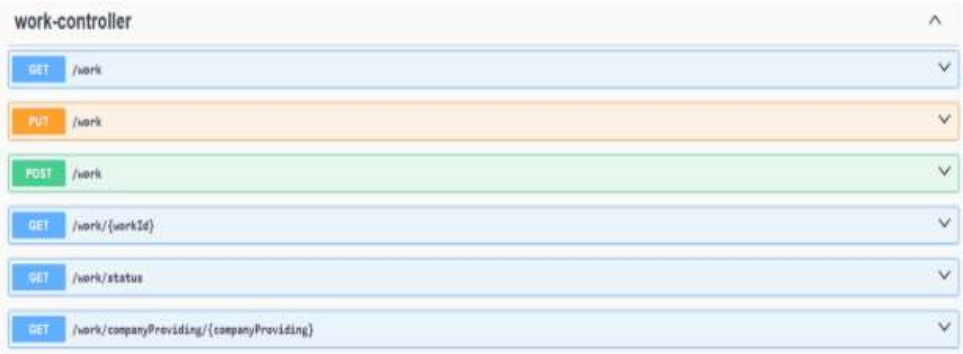
As Figuras 8 a 15 demonstram a documentação gerada:

Figura 8: Interface Principal



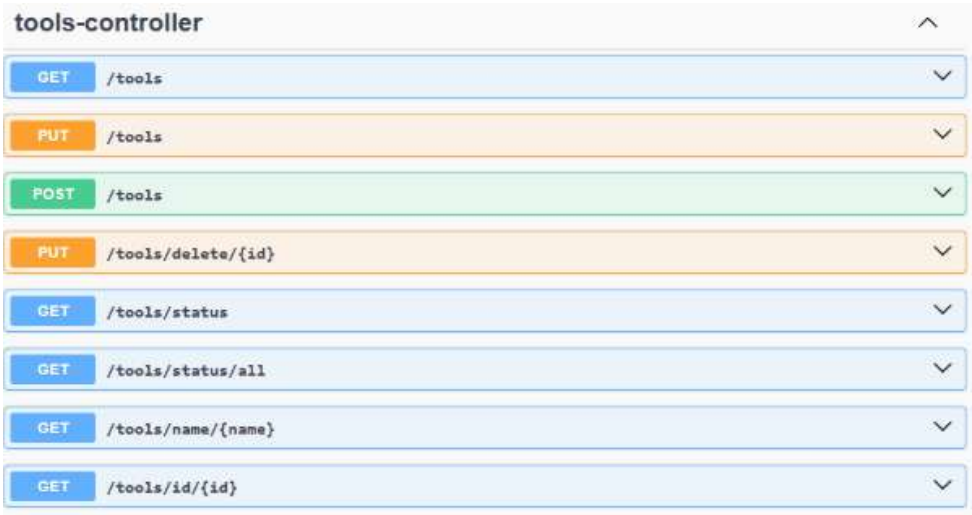
Fonte: Próprio Autor.

Figura 9: Endpoint de Work



Fonte: Próprio Autor.

Figura 10: Endpoint de Tools

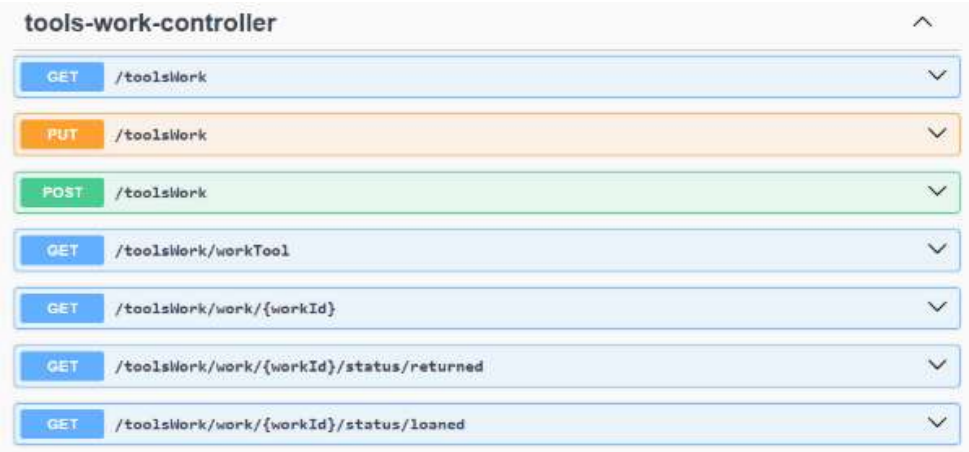


Fonte: Próprio Autor.

Figura 11: Endpoint de Employee-Work

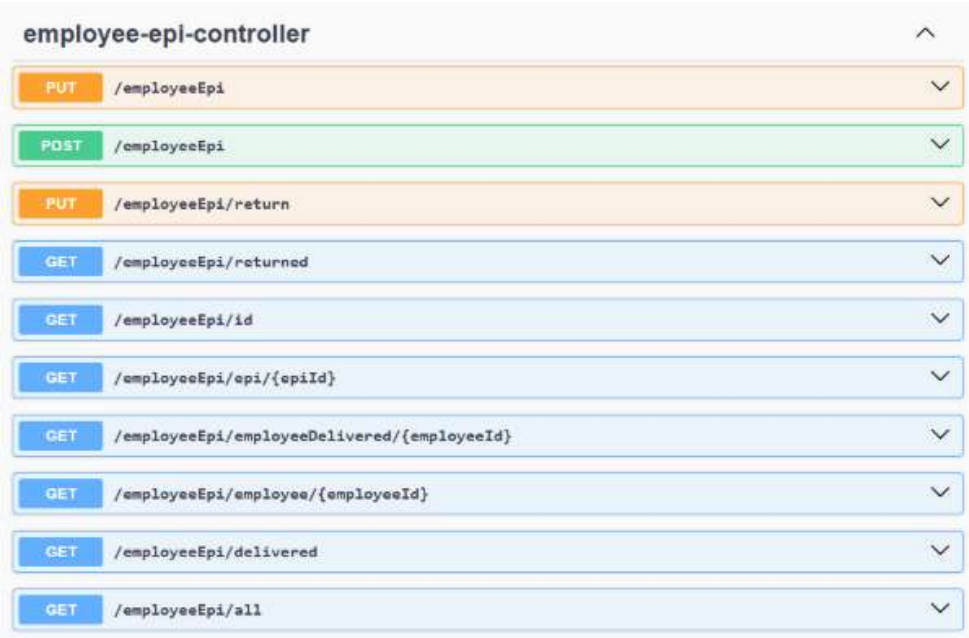
Fonte: Próprio Autor.

Figura 12: Endpoint de Tools-Work



Fonte: Próprio Autor.

Figura 13: Endpoint de Employee-Epi

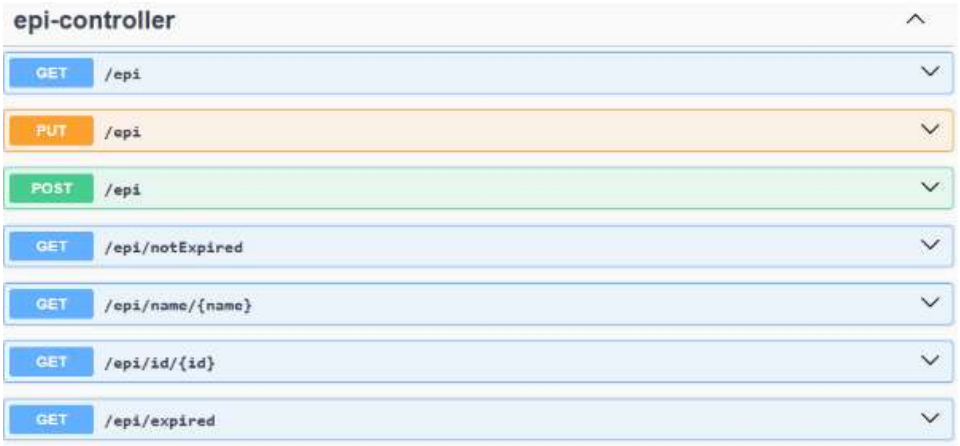


Fonte: Próprio Autor.

Figura 14: Endpoint de Employee-Tool

Fonte: Próprio Autor.

Figura 15: Endpoint de EPI

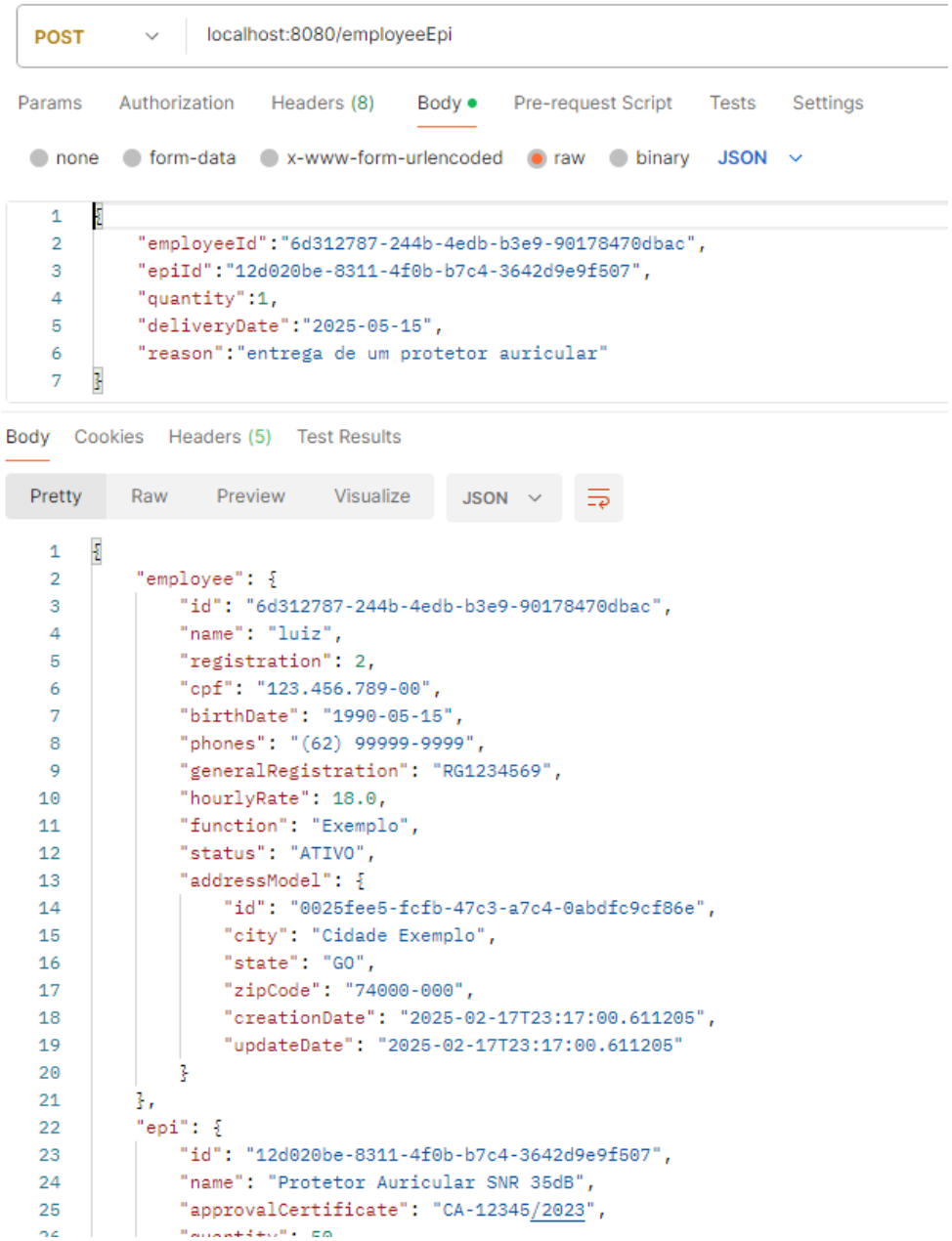


Fonte: Próprio Autor.

3.9.2 Utilização da API

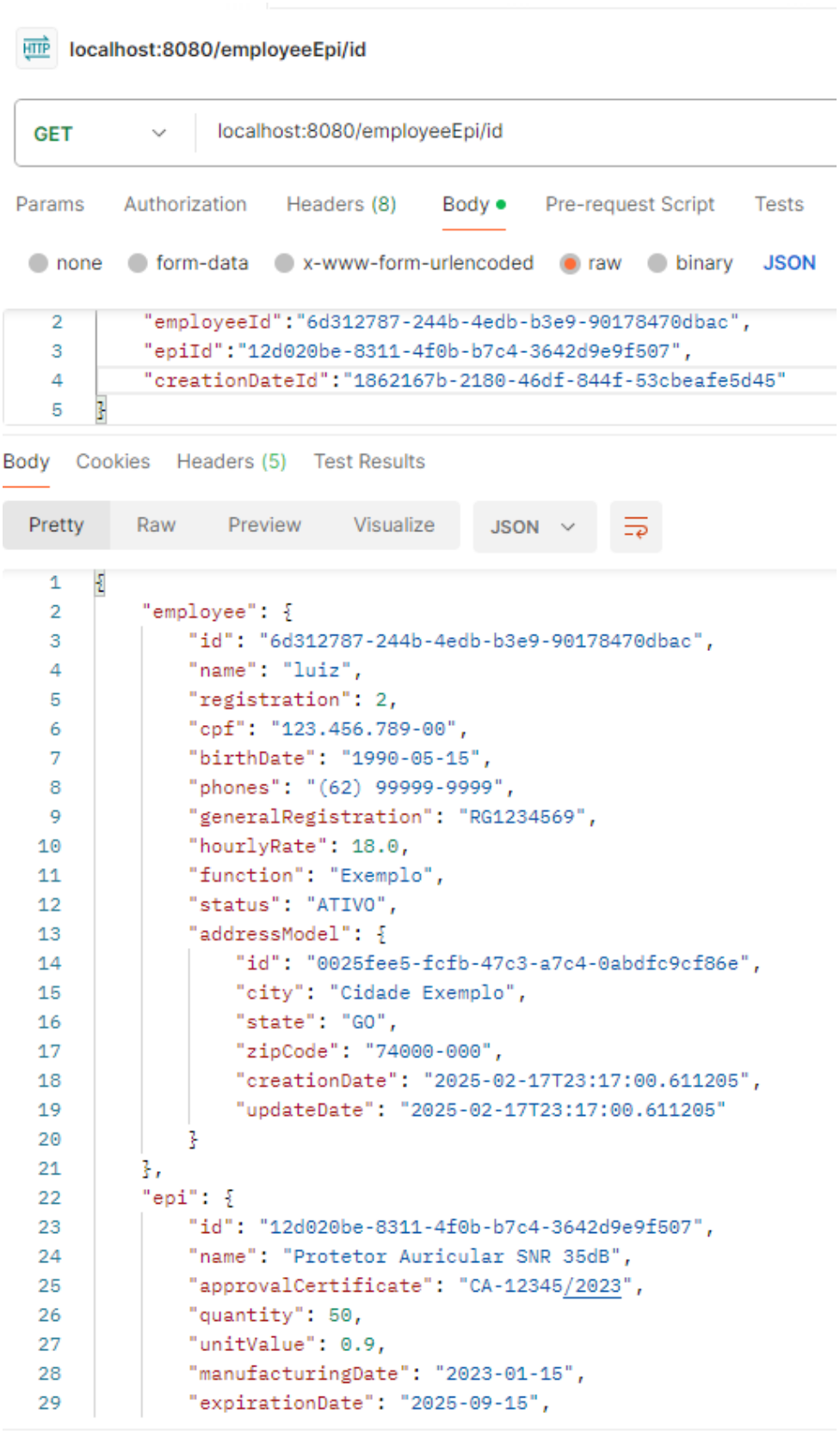
A API foi projetada seguindo as práticas REST, conforme definido por Fielding (2000), com enfoque na simplicidade e consistência das operações. Em seguida será exibido duas imagens que representa o cadastro e listagem de um EPI sendo entregue a um funcionário.

Figura 16: Cadastro de EmployeeEpi



Fonte: Próprio Autor.

Figura 17: Listagem de EmployeeEpi especifico



Fonte: Próprio Autor.

4 CONCLUSÃO

O desenvolvimento da API RENOVAR atendeu plenamente aos objetivos propostos neste trabalho. A solução criada permite o gerenciamento integrado de funcionários, ferramentas e Equipamentos de Proteção Individual (EPIs), conforme definido no objetivo geral. Por meio de uma arquitetura moderna utilizando Kotlin, Spring Boot, PostgreSQL e Docker, foi possível entregar uma aplicação robusta e escalável.

Em relação aos objetivos específicos, a gestão de funcionários foi facilitada por funcionalidades como cadastro, atualização e visualização de dados, além da associação de EPIs e ferramentas a cada colaborador. O monitoramento do estado e disponibilidade dos equipamentos foi implementado por meio de endpoints específicos que permitem o controle de estoque, situação de uso e histórico de movimentações. A integração entre os módulos e sistemas externos foi garantida pela implementação de uma API REST bem estruturada e documentada com Swagger UI.

Os testes realizados demonstraram a confiabilidade da solução, com 80% de cobertura de linhas e 89% de branches, o que reforça a qualidade do código e a aderência às boas práticas de desenvolvimento. Com isso, conclui-se que todos os objetivos definidos foram atingidos de forma eficaz, resultando em um sistema funcional, documentado e preparado para evoluções futuras.

REFERÊNCIAS

Af Figueiredo. *Ausência de EPIs ocasiona indenização por danos morais*. 2018. Acesso: 23 mar. 2025. Disponível em: <https://www.affigueiredo.com.br/2018-05-29-ausencia-de-epis-ocasiona-indenizacao-por-danos-morais/>.

Associação Nacional de Medicina do Trabalho. *Construção civil está entre os setores com maior risco de acidentes de trabalho*. 2019. Acesso: 23 mar. 2025. Disponível em: <https://www.anamt.org.br/portal/2019/04/30/construcao-civil-esta-entre-os-setores-com-maior-risco-de-acidentes-de-trabalho/>.

BITTERNCOURT, M. D. *Testes Unitários com JUnit 5*. Leanpub, 2020. Acesso: 05 mai. 2025. Disponível em: <https://leanpub.com/testesunitarioscomjunit5>.

Docker Inc. *Docker Documentation*. 2023. Acesso: 05 mai. 2025. Disponível em: <https://docs.docker.com/>.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. Acesso: 05 mai. 2025. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

INITIATIVE, O. Openapi specification v3.1.0. 2023. Acesso: 20 mai. 2025. Disponível em: <https://spec.openapis.org/oas/v3.1.0>.

MARMSOLER, D.; MALKIS, A.; ECKHARDT, J. A model of layered architectures. *arXiv preprint*, 2015. Acesso: 01 mai. 2025. Disponível em: <https://arxiv.org/abs/1503.04916>.

MYERS, G. J. *The Art of Software Testing*. 2. ed. New York: Wiley, 2004.

SANTOSS, W.; DIEGUES, U. C. C. A importância da língua estrangeira para o estudante de análise e desenvolvimento de sistemas. *Revista Processando o Saber*, v. 13, p. 267–277, 2021.

SHVETS, A. *Mergulho nos Padrões de Projeto*. Espanha: [s.n.], 2021.

Sintricomb. *Estudo mostra que 40% dos acidentes de trabalho no Brasil são por queda de altura*. 2019. Acesso: 23 mar. 2025. Disponível em: <https://sintricomb.com.br/estudo-mostra-que-40-dos-acidentes-de-trabalho-no-brasil-sao-por-queda-de-altura/>.

SMITH, J.; JOHNSON, M. *API Documentation Best Practices*. 2nd. ed. [S.l.]: Tech Publications, 2022.

SOMMERVILLE, I. *Engenharia de Software*. 9. ed. São Paulo: Pearson, 2011.

TREVISAN, F. C. Trabalho de Conclusão de Curso (Graduação em Engenharia Civil), *Análise das condições de segurança do trabalho em canteiros de obras conforme NR 18 no município de Santa Cruz do Sul*. Porto Alegre: [s.n.], 2015. Acesso: 23 mar. 2025. Disponível em: <https://lume.ufrgs.br/handle/10183/138289>.