Aminah Roca Makhoul

Desenvolvimento de API para Gerenciamento de Locação de Ferramentas

Vilhena - RO

2025

Aminah Roca Makhoul

Desenvolvimento de API para Gerenciamento de Locação de Ferramentas

Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de Rondônia – campus Vilhena, realizado em cumprimento de requisito parcial para a obtenção do título de Tecnóloga em Análise e Desenvolvimento de Sistemas.

Instituto Federal de Educação, Ciência e Tecnologia de Rondônia – IFRO

Campus Vilhena

Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas

Orientador: Prof. MSc. Gilberto Pereira da Silva

Vilhena - RO

2025

Ficha catalográfica elaborada pelo Sistema Gerador de Ficha Catalográfica do IFRO, com dados informados pelo(a) próprio(a) autor(a).

Makhoul, Aminah Roca.
Desenvolvimento de API para gerenciamento de locação de ferramentas / Aminah Roca Makhoul, Vilhena-RO, 2025.
66 f.

Orientador(a): Prof. Ms. Gilberto Pereira da Silva.

Trabalho de Conclusão de Curso (Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO, Vilhena-RO, 2025.

1. API. 2. Locação. 3. Kanban. I. da Silva, Gilberto Pereira (orient.). II. Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO. III. Título.

Bibliotecário(a) Responsável: Rosilene Maria do Couto Marques, CRB-11/321 (Campus Vilhena)

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE RONDÔNIA
Vilhena - Código INEP: 11107804
Rodovia BR 174, KM 3, CEP 76982-270, Vilhena (RO)
CNPJ: 10.817.343/0003-69 - Telefone: 69 2101-0703

ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO

Na data 08/04/2025 realizou-se a sessão pública de defesa do Trabalho de Conclusão de Curso intitulada **Api para o gerenciamento de locação de ferramentas** apresentada pela aluna **Aminah Roca Makhoul (2022103070020)** do Curso **Superior de Tecnologia em Análise e Desenvolvimento de Sistemas (Vilhena)**. Os trabalhos foram iniciados às **16:00** pelo Professor **Gilberto Pereira da Silva** presidente da banca examinadora, constituída pelos seguintes membros:

- **Gilberto Pereira da Silva** (Orientador)
- **Roberto Simplicio Guimaraes** (Examinador Interno)
- **Wesley Jhonnes Ramos Rolim** (Examinador Interno)

A banca examinadora, tendo terminado a apresentação do conteúdo do Trabalho de Conclusão de Curso, passou à arguição da candidata. Em seguida, os examinadores reuniram-se para avaliação e deram o parecer final sobre o trabalho apresentado pelo aluno, tendo sido atribuído o seguinte resultado:

[X] APROVADO **Nota: 93**

Proclamados os resultados pelo presidente da banca examinadora, foram encerrados os trabalhos e, para constar, eu **Gilberto Pereira da Silva** lavrei a presente ata que assino juntamente com os demais membros da banca examinadora.

VILHENA / RO, 08/04/2025

Documento assinado eletronicamente por **Aminah Roca Makhoul**, Discente, em 08/04/2025, às 18:04, conforme horário oficial de Rondônia, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Gilberto Pereira da Silva**, Orientador, em 08/04/2025, às 17:09, conforme horário oficial de Rondônia, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Roberto Simplicio Guimaraes**, Examinador Interno, em 08/04/2025, às 17:09, conforme horário oficial de Rondônia, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Wesley Jhonnes Ramos Rolim**, Examinador Interno, em 08/04/2025, às 17:09, conforme horário oficial de Rondônia, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

“Sua única limitação é aquela que você impõe em sua própria mente.”
Napoleon Hill

Agradecimentos

Agradeço ao professor orientador Gilberto Pereira da Silva, pela dedicação, confiança e paciência ao longo de todo o processo de desenvolvimento deste trabalho e a todos os professores que contribuíram para minha formação ao longo do curso. A ajuda deles foi fundamental para a elaboração deste TCC. Um agradecimento especial à professora Rosa Maria da Silva Gonçalves e ao professor Roberto Simplicio Guimarães, que ofereceram suporte valioso e dicas importantes, além das correções construtivas que aprimoraram a qualidade desta monografia.

Gostaria de expressar minha profunda gratidão àqueles que sempre estiveram ao meu lado, oferecendo apoio emocional e motivacional ao longo desta jornada. Sem esse suporte constante, seria muito mais difícil seguir em frente. A dedicação e a confiança dessas pessoas foram essenciais para que eu conseguisse concluir este trabalho.

Por fim, agradeço aos meus colegas de sala, que tornaram esta jornada acadêmica muito melhor com ajuda e espírito colaborativo. E a todos que não foram mencionados, mas que ajudaram no processo deste trabalho, o meu muito obrigado!

Resumo

Este trabalho desenvolve uma API *(Application Programming Interface* - Interface de Programação de Aplicações) para gerenciar locações de ferramentas em pequenas empresas. Muitas dessas empresas enfrentam dificuldades com o controle manual, o que pode resultar em erros e prejuízos financeiros. O objetivo é criar um sistema automatizado para melhorar a precisão e a eficiência no gerenciamento das locações. Utilizando tecnologias como *JavaScript*, *NodeJs, Docker, MongoDB e ExpressJs*, o projeto emprega a metodologia *Kanban* para garantir um desenvolvimento ágil e eficaz. A API facilitará o controle de entrada e saída de ferramentas, reduzindo riscos e aumentando a satisfação dos administradores.

Palavras-chave: API; *JavaScript; Node.js; Docker; MongoDB; Express; Kanban;* gerenciamento de locações.

Abstract

This work develops an API (*Application Programming Interface*) for managing tool rentals in small businesses. Many of these businesses face challenges with manual control, which can lead to errors and financial losses. The goal is to create an automated system to improve accuracy and efficiency in rental management. Using technologies such as *JavaScript, NodeJS, Docker, MongoDB, and ExpressJS*, the project employs the *Kanban* methodology to ensure agile and effective development. The API will facilitate the control of tool check-ins and check-outs, reducing risks and increasing administrator satisfaction.

Keywords: API; *JavaScript; Node.js; Docker; MongoDB; Express; Kanban*; *rental management.*

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
CRUD	Create (criação), Read (consulta), Update (atualização) e Delete (deleção) de dados
DOM	Document Object Model
ERP	Planejamento de Recursos Empresariais
GIT	Git Version Control System
HTTP	HyperText Transfer Protocol
ID	Identificador
JIT	Just-in-Time
JS	JavaScript
JSON	JavaScript Object Notation
NoSQL	Not Only SQL (não apenas SQL)
ODM	Object Data Modeling
ORM	Object Relational Mapping
RESTful	Representational State Transfer
RF	Requisitos Funcionais
RNF	Requisitos Não Funcionais
RQ	Requisitos Não Funcionais
SGBD	Sistema Gerenciador de Banco de Dados
SGBDR	Sistema Gerenciador de Banco de Dados Relacional
UML	Unified Modeling Language
VM	Virtual Machine

Sumário

1 Introdução

1.1 Contexto e problema

A administração eficaz de locações de ferramentas é fundamental para o bom funcionamento de pequenas empresas que oferecem serviços de locação. No entanto, muitas dessas empresas ainda utilizam processos manuais para gerenciar a entrada e saída de ferramentas, o que gera diversos problemas de gestão. Métodos tradicionais, como registros em papel ou planilhas, frequentemente resultam em erros de inserção de dados, perda de informações e, em alguns casos, prejuízos financeiros significativos (LAUDON; LAUDON, 2019).

Além disso, a gestão manual demanda muito tempo e é suscetível a falhas humanas, o que compromete a eficiência operacional e afeta tanto a satisfação dos clientes quanto a precisão dos registros. A ausência de um sistema automatizado não apenas eleva os custos administrativos, mas também limita a capacidade da empresa de monitorar e controlar suas operações de maneira eficaz (TURBAN et al., 2018).

Diante desse cenário, observa-se um problema: a falta de uma solução tecnológica adequada para a gestão de locações de ferramentas. Sem um sistema eficiente, pequenas empresas enfrentam dificuldades constantes na garantia de precisão e eficiência na administração de seus recursos. Assim, a implementação de uma solução automatizada torna-se essencial para superar esses desafios e otimizar as operações empresariais.

Nesse contexto, surge o questionamento: de que maneira uma aplicação para gerenciamento de locações de ferramentas pode aprimorar o controle e a emissão de registros, assegurando precisão e eficiência na administração dos recursos?

1.2 Objetivos

1.2.1 Objetivo geral

Desenvolver uma API *RESTful* para o gerenciamento de locações de ferramentas.

1.2.2 Objetivos específicos

Para alcançar o objetivo geral descrito acima, serão seguidos os seguintes passos, os quais constituem os objetivos específicos:

- Levantar e documentar os requisitos funcionais e não funcionais da API, abrangendo operações de cadastro, consulta, atualização e exclusão de dados, além de requisitos de desempenho, segurança, disponibilidade e documentação.
- Desenvolver a API com documentação completa, incluindo especificações técnicas, endpoints, métodos HTTP, estruturas de dados e regras de validação.
- Configurar e implementar um banco de dados utilizando *MongoDB* para armazenar e gerenciar as informações com segurança e eficiência.
- Realizar testes automatizados para validar a funcionalidade, segurança e desempenho da API.
- Preparar e disponibilizar a API em ambiente de produção utilizando *Docker*.

1.3 Justificativa

Esta API oferece uma solução tecnológica moderna e eficiente voltada para pequenas empresas, com o propósito de centralizar e automatizar o controle de entrada e saída de ferramentas alugadas, substituindo métodos manuais tradicionais, como registros em papel ou planilhas, que são suscetíveis a erros humanos e à perda de informações (LAUDON; LAUDON, 2019).

Ao adotar essa solução tecnológica, as empresas poderão gerenciar seus recursos de maneira mais organizada e eficiente, minimizando o risco de perdas e prejuízos financeiros decorrentes de falhas na gestão manual. Além disso, a automação dos processos proporcionará maior precisão na emissão de registros, permitindo um acompanhamento detalhado das locações e devoluções, o que contribui para uma melhor tomada de decisões estratégicas.

Outro benefício significativo é a otimização do tempo e a redução de custos operacionais, uma vez que o sistema automatizado elimina tarefas repetitivas e manuais, permitindo que os colaboradores se concentrem em atividades mais relevantes para o crescimento da empresa. Com isso, a API não apenas melhora a eficiência operacional, mas também aumenta a satisfação dos clientes ao oferecer um serviço mais ágil e confiável.

Portanto, o desenvolvimento desta API visa transformar a maneira como pequenas empresas gerenciam suas locações de ferramentas, proporcionando um sistema inovador que combina praticidade, segurança e eficiência. Dessa forma, espera-se não apenas modernizar a gestão de recursos, mas também fortalecer a competitividade dessas empresas no mercado.

2 Fundamentação teórica

2.1 Trabalhos similares

No decorrer da pesquisa e desenvolvimento deste projeto, não foram encontradas alternativas de código aberto que contemplassem os mesmos objetivos aqui propostos. As opções identificadas se restringiram a softwares proprietários, dentre os quais se destacam:

- *RentHub*[1]: O *RentHub* é um *software* de gestão de locações baseado na nuvem, desenvolvido para automatizar atividades de aluguel. Apesar de ser uma solução paga, é possível solicitar uma demonstração gratuita. A plataforma oferece três soluções especializadas: aluguel de carros, aluguel de equipamentos e aluguel de veículos com motorista. Isso permite que os clientes escolham o *software* mais adequado às suas necessidades.

- *Brascomm*[2]: O ERP(planejamento de recursos empresariais) *Brascomm* 4.0 é uma solução de sistema de gestão projetada para empresas do segmento de locação de equipamentos. Este sistema não é gratuito e compreende as necessidades administrativas de diversos nichos, como Locação de Máquinas de Café, Equipamentos Hospitalares, Equipamentos de Construção Civil e Empilhadeiras. O ERP *Brascomm* 4.0 oferece uma infraestrutura completa, proporcionando uma gestão organizada, segura e adaptada às particularidades do segmento de locação.

- *Sisloc*[3]: O *Sisloc* é um *software* de gestão amplamente reconhecido no Brasil, desenvolvido para locadoras de equipamentos e é um *software* pago. O *Sisloc* otimiza os processos de locação de bens móveis, ajudando as empresas a reduzir custos e alcançar melhores resultados por meio de soluções tecnológicas avançadas.

2.2 Interface de Programação de Aplicações

API é um acrônimo para *Application Programming Interface* (Interface de Programação de Aplicação). De acordo com (JIN; SAHNI; SHEVAT, 2018), as APIs desempenham um papel essencial na integração entre sistemas, permitindo que diferentes aplicações se comuniquem de maneira estruturada e eficiente. Elas estabelecem um conjunto de regras que definem como os componentes de software devem interagir, garantindo padronização e reutilização de funcionalidades. No contexto do desenvolvimento moderno, as APIs

[1] Disponível em https://www.renthubsoftware.com/pt/
[2] Disponível em https://www.brascomm.net.br/
[3] Disponível em https://sisloc.com/

possibilitam a criação de sistemas modulares, nos quais aplicações independentes podem compartilhar dados e serviços sem a necessidade de implementações redundantes. Esse conceito é amplamente adotado em serviços web, onde APIs permitem a interoperabilidade entre plataformas distintas, promovendo escalabilidade e flexibilidade no desenvolvimento de software.

2.3 Banco de Dados NoSQL

MongoDB é um banco de dados orientado a documentos, desenvolvido para oferecer maior escalabilidade em comparação aos bancos relacionais, conforme destacado por (CHODOROW, 2013). Em vez de utilizar o modelo tradicional baseado em tabelas e linhas, ele adota documentos flexíveis que permitem a inclusão de dados aninhados e *arrays*. Essa estrutura possibilita representar relações hierárquicas complexas dentro de um único registro, o que se alinha ao modo como desenvolvedores de linguagens orientadas a objetos organizam seus dados. Em comparação aos SGBDR(Sistema Gerenciador de Banco de Dados Relacional) algumas analogias podem ser feitas: uma tabela pode ser comparada com uma coleção e um documento pode ser comparado a uma linha de dados (SCHROEDER; SANTOS,).

2.4 Linguagem de Programação JavaScript

JavaScript é a linguagem de programação da Web (comumente abreviado para JS). A ampla maioria dos sites modernos usa *JavaScript* e todos os navegadores modernos – em computadores de mesa, consoles de jogos, *tablets* e *smartphones* – incluem interpretadores *JavaScript*, tornando-a a linguagem de programação mais onipresente da história (FLANAGAN, 2012). Ela permite a criação de interações dinâmicas em páginas da web, como validações de formulário, animações e a manipulação de elementos do DOM (*Document Object Model*). Desde seu lançamento em 1995, *JavaScript* evoluiu significativamente, tornando-se essencial para o desenvolvimento tanto no *front-end* quanto no *back-end* através de *frameworks* e bibliotecas populares como *Node.js*. Além disso, sua flexibilidade e suporte a diversos paradigmas de programação, incluindo orientação a objetos e funcional, têm contribuído para sua popularidade contínua entre desenvolvedores (Mozilla Developer Network, 2023).

2.5 Metodologia kanban

O desenvolvimento desta aplicação seguiu os princípios da metodologia *Kanban*, originada no Sistema Toyota de Produção no final da década de 1940. Naquele período, a Toyota implementou o método *Just-in-Time* (JIT), que difere do modelo de produção em massa desenvolvido por Henry Ford, no qual os produtos eram produzidos em grandes

quantidades para depois serem enviados ao mercado. O JIT, por sua vez, adota um sistema "puxado", onde a produção é guiada pela demanda do mercado, o que minimiza desperdícios e melhora a eficiência do tempo de produção (KANBANIZE, 2024).

A palavra "*Kanban*", de origem japonesa, se traduz como "cartão de sinalização". Em um sistema *Kanban*, as tarefas geralmente são organizadas em três colunas: “Pedido”, “Em progresso” e “Concluído”. Quando gerido adequadamente, esse sistema oferece uma visão clara e em tempo real do processo de trabalho, ajudando a identificar gargalos e obstáculos que possam prejudicar o desempenho. A metodologia *Kanban* também está intimamente ligada ao processo de melhoria contínua (*Kaizen*) adotado pela Toyota, sendo essencial para o sucesso dessa abordagem.

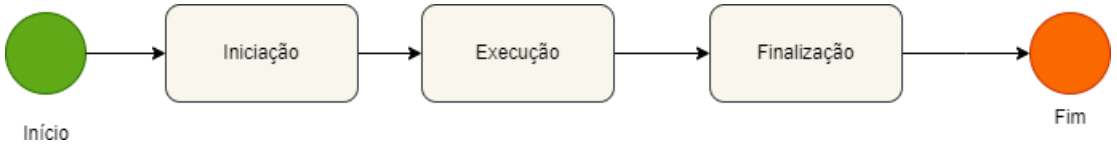
Quando aplicada ao desenvolvimento de software, o sistema *Kanban* virtual é utilizado para controlar o trabalho em progresso, limitando a quantidade de tarefas executadas simultaneamente. Embora os cartões em *Kanban* sejam tradicionalmente usados para indicar a necessidade de mais trabalho, no contexto do *software*, eles servem para ilustrar as tarefas que estão em andamento. A indicação para iniciar uma nova tarefa é calculada com base no número de tarefas já em execução, subtraindo o limite previamente estabelecido. Essa metodologia visa equilibrar a carga de trabalho da equipe de desenvolvimento com a capacidade de entrega, criando um ritmo de trabalho sustentável. Limitar o número de tarefas em andamento não só melhora a eficiência, mas também contribui para a qualidade do trabalho, ao permitir que a equipe foque nas tarefas em mãos. Como resultado, a melhoria contínua do fluxo de trabalho e a maior qualidade do produto ajudam a reduzir o tempo de entrega, promovendo uma cadência constante e previsível de entregas (ANDERSON; REINERTSEN; PINTO, 2011).

3 Materiais e Métodos

3.1 Ciclo de Vida do Desenvolvimento da API

O ciclo de vida do desenvolvimento da API é composto por três fases principais mostradas na Figura 1: Iniciação, Execução e Finalização. Cada fase contém subprocessos específicos, os quais são fundamentais para a construção de uma aplicação sólida e eficiente. O processo seguiu o modelo incremental, que é ideal para situações onde a primeira entrega consiste em uma versão funcional, porém ainda não completamente finalizada, mas com funcionalidades essenciais já utilizáveis (PRESSMAN; MAXIM, 2016).

Figura 1 – Ciclo de Vida



Fonte: elaborado pelo autor (2025).

A fase de iniciação, apresentada na figura 2, marca o início do projeto, quando ele começou a tomar forma. Nessa etapa, foi essencial compreender as necessidades do sistema, definir o escopo e planejar as ações iniciais. A análise de requisitos desempenhou um papel crucial para garantir uma compreensão precisa das expectativas do cliente e das funcionalidades a serem implementadas. O objetivo foi assegurar clareza e alinhamento desde o início.

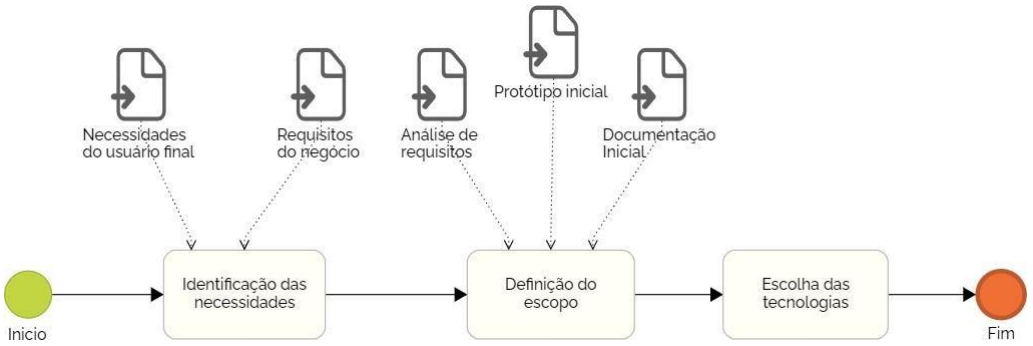
Além disso, nesse período, ocorreu a escolha das tecnologias a serem adotadas. Para o desenvolvimento da API, optou-se por utilizar *JavaScript* [1], *NodeJS* [2], *ExpressJS* [3] e *MongoDB* [4], pois essas ferramentas atendem às exigências do projeto em termos de desempenho e flexibilidade. *NodeJS* se destaca por sua capacidade de garantir execução rápida e escalável do código, enquanto *ExpressJS* facilita a organização das rotas da API. *MongoDB* foi selecionado por ser um banco de dados não relacional, que se adapta facilmente ao crescimento da aplicação, sem exigir grandes alterações estruturais.

Em resumo, a fase de iniciação envolveu decisões fundamentais, como a compreensão das necessidades do sistema e o planejamento adequado para garantir um desenvolvimento

[1] Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript
[2] Disponível em: https://nodejs.org/en/about/
[3] Disponível em: https://expressjs.com/
[4] Disponível em: https://www.mongodb.com/what-is-mongodb

eficiente e organizado. Essa fase estabeleceu a base para o restante do projeto, assegurando uma visão clara das ações necessárias e a abordagem a ser seguida.

Figura 2 – Fase de Iniciação



Fonte: elaborado pelo autor (2025).

Na fase de execução, apresentada na figura 3, aconteceu a implementação prática de tudo o que foi planejado na fase anterior. Esse foi o momento em que as ideias se transformaram em realidade, com foco no desenvolvimento do sistema. A principal meta foi garantir a execução eficiente de todas as partes do projeto, atendendo aos requisitos e funcionalidades definidos previamente.

Iniciou-se a codificação, que consiste no processo de escrever o código-fonte. Utilizaram-se as tecnologias selecionadas, como *NodeJS*, *ExpressJS* e *MongoDB*, com o *JavaScript* como linguagem principal. Durante essa etapa, seguiram-se boas práticas de programação para assegurar a clareza do código e facilitar sua compreensão, o que é essencial para a manutenção futura.

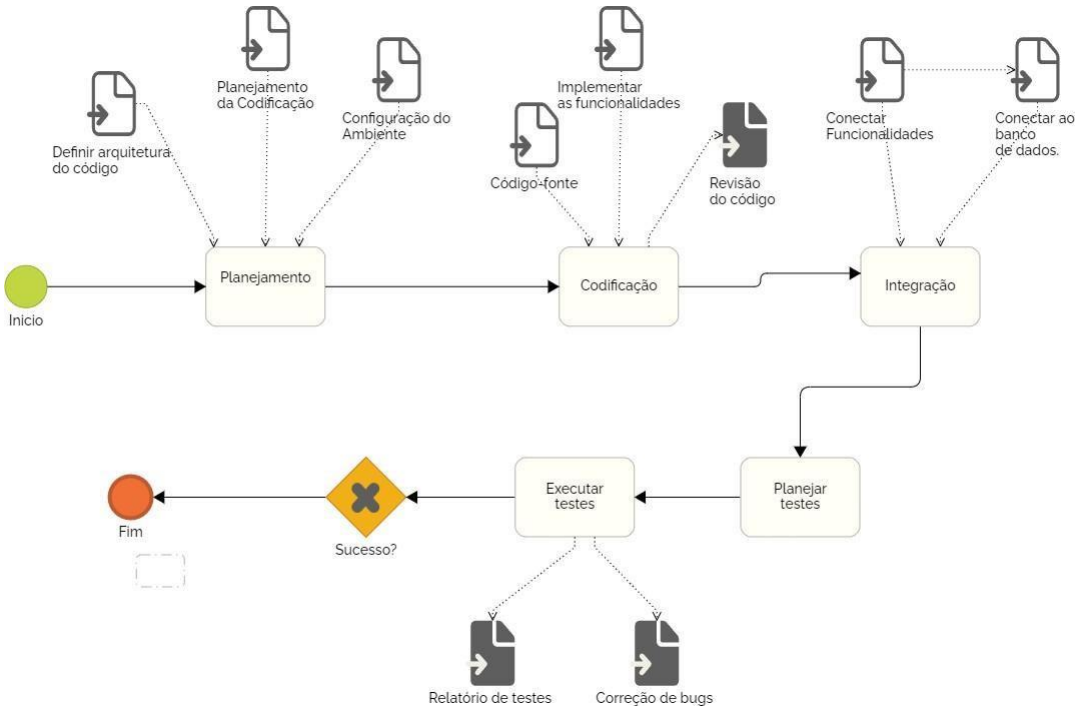
Além disso, foram realizados testes manuais para verificar se o sistema estava funcionando conforme o planejado. Esse processo permitiu a identificação de erros e sua correção antes da integração completa. A integração das funcionalidades foi uma etapa crucial, pois envolveu a conexão das diversas partes do sistema, como o banco de dados e os diferentes módulos da aplicação, assegurando que todas as partes funcionassem de forma integrada e sem falhas.

Monitorar o progresso foi essencial. Ajustes ocorreram no código, nas funcionalidades e no cronograma, sempre com o objetivo de manter o projeto no caminho certo. O monitoramento contínuo permitiu perceber qualquer desvio do plano e realizar correções de forma ágil.

Em resumo, a fase de execução transformou a visão do projeto em uma solução real e funcional. Com a codificação e os ajustes realizados, foi possível criar o sistema

conforme o planejamento, garantindo a entrega de um produto final pronto para uso.

Figura 3 – Fase de Execução



Fonte: elaborado pelo autor (2025).

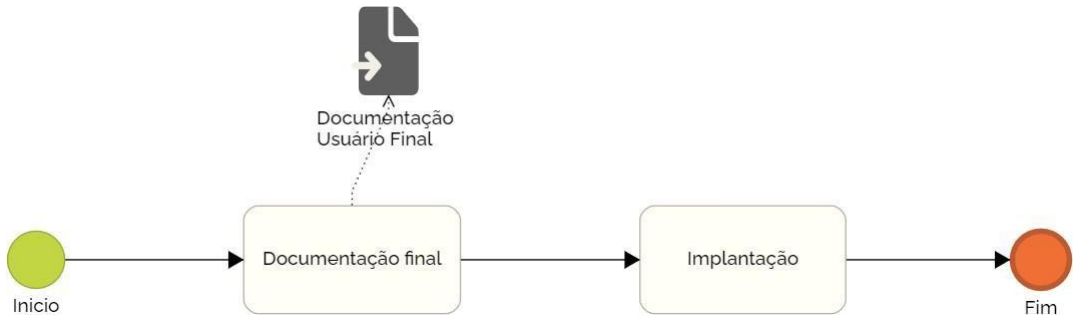
A fase de finalização, apresentada na figura 4, representa a última etapa do projeto, quando tudo foi revisado, testado e preparado para entrega. Nessa fase, realizei testes de unidade e de integração para garantir o funcionamento adequado de todos os módulos desenvolvidos, de forma integrada e sem falhas. Em seguida, ocorreu a validação do sistema, etapa essencial para confirmar se ele atende aos requisitos definidos na fase inicial, simulando o uso real em ambiente controlado.

Com tudo validado, fiz a entrega do produto, disponibilizando a aplicação para o ambiente de produção, assegurando que estivesse pronta para uso. Durante essa entrega, forneci instruções de utilização para facilitar a adoção do sistema.

Além disso, finalizei a documentação técnica e funcional, que abrange informações detalhadas sobre o sistema, configurações, funcionalidades e instruções para manutenção futura. Por fim, fiz uma revisão geral e o encerramento do projeto, com a apresentação final e a confirmação de que todas as entregas planejadas foram cumpridas com sucesso.

Essa fase conclui o ciclo do projeto, garantindo o funcionamento do sistema conforme as expectativas e a prontidão para a operação.

Figura 4 – Fase de Finalização



Fonte: elaborado pelo autor (2025).

3.2 Ferramentas e Tecnologias Utilizadas

3.2.1 NodeJS

De acordo com (NGUYEN, 2012), o *Node.js* é uma plataforma de execução *JavaScript* orientada ao lado do servidor, caracterizada por uma biblioteca central minimalista, mas altamente funcional. Ao contrário de outras plataformas, que podem ser mais complexas e pesadas, o *Node.js* adota uma abordagem simplificada, permitindo que desenvolvedores integrem facilmente bibliotecas adicionais conforme necessário.

Construído sobre o motor V8, o mesmo utilizado pelo Google Chrome, o *Node.js* se beneficia de alta velocidade e eficiência na execução de código *JavaScript*. Graças a esse motor de alto desempenho, o *Node.js* consegue processar grandes volumes de dados de maneira rápida, tornando-se ideal para aplicações em tempo real, como chats, jogos online e servidores de alta performance (NGUYEN, 2012).

O *Node.js* foi utilizado como ambiente de execução no *back-end*, permitindo a execução de código *JavaScript* no servidor. Sua arquitetura assíncrona e a capacidade de lidar com um grande número de requisições simultâneas tornam-no uma escolha ideal para esses cenários.

3.2.2 ExpressJS

O *ExpressJS* [5] é um *framework* para criar aplicações *web*, que facilita o trabalho com *Node.js*. Ele ajuda a criar APIs de forma rápida e fácil, fornecendo recursos essenciais para o desenvolvimento. Uma das vantagens do *ExpressJS* é o uso de *"middleware"*, que permite adicionar funcionalidades extras conforme a necessidade (EXPRESSJS, 2025).

[5] Disponível em: https://expressjs.com/

O *ExpressJS* foi escolhido como o *framework* principal para organizar as rotas da *API*, permitindo que as diferentes funcionalidades do sistema fossem estruturadas de forma clara e eficiente.

3.2.3 MongoDB

O *MongoDB* [6] foi escolhido para armazenar as informações da aplicação devido à sua flexibilidade e capacidade de adaptação ao crescimento do sistema. A adaptação ocorre porque o *MongoDB* é um banco de dados NoSQL, ou seja, não-relacional, e utiliza documentos BSON (semelhantes ao JSON). Isso significa que, à medida que a aplicação cresce e novas funcionalidades são adicionadas, podemos modificar facilmente a estrutura dos dados, incluindo novos campos ou coleções, sem a necessidade de reestruturar todo o banco de dados, como ocorre em bancos relacionais.

Por exemplo, se um novo recurso for adicionado ao sistema, como a necessidade de armazenar informações extras sobre as locações e ferramentas, o *MongoDB* pode ser facilmente ajustado para incluir esses novos dados. Basta adicionar novos campos aos documentos existentes, sem a necessidade de reestruturar a base de dados inteira, o que é comum em bancos relacionais.

Essa flexibilidade é crucial porque, à medida que a aplicação cresce e novas funcionalidades são implementadas, a estrutura de dados pode ser modificada sem grandes complicações ou impactos no desempenho do sistema.

Assim, o *MongoDB* permite uma adaptação contínua ao crescimento do sistema, com a possibilidade de expandir funcionalidades e dados de maneira simples e eficiente, sem comprometer a estrutura já existente.

Portanto, o uso dele foi essencial para a flexibilidade do sistema, permitindo que novas funcionalidades fossem integradas sem grandes modificações na estrutura de armazenamento.

3.2.3.1 Mongoose

O *Mongoose* [7] é uma ferramenta essencial para quem trabalha com *MongoDB* em conjunto com *Node.js*. Sua principal função é fornecer um modelo de dados estruturado, facilitando a interação com os dados no banco de dados. De acordo com (HOLMES, 2013), ao usar o *Mongoose*, o desenvolvedor pode definir esquemas, garantindo que os dados sejam armazenados de forma organizada e consistente. Isso simplifica as operações em *MongoDB*, pois o *Mongoose* lida com tarefas comuns, como validação e manipulação de dados, de forma eficiente.

[6] Disponível em: https://www.mongodb.com/what-is-mongodb

[7] Disponível em: https://www.mongodb.com/developer/languages/javascript/getting-started-with-mongodb-and-mongoose/

Além disso, o *Mongoose* resolve parte da complexidade encontrada ao trabalhar diretamente com o driver nativo do *MongoDB,* como o uso de *callbacks* aninhados. Ele oferece métodos que tornam o código mais limpo.

Neste projeto, a ferramenta foi utilizada para tornar a API mais eficiente, organizada e fácil de manter.

3.2.4 UML

A *UML* - (*Unified Modeling Language*) ou Linguagem de Modelagem Unificada, é uma linguagem visual utilizada para modelar softwares baseados no paradigma de orientação a objetos. É uma linguagem de modelagem de propósito geral que pode ser aplicada a todos os domínios de aplicação. Essa linguagem é, atualmente, a linguagem-padrão de modelagem adotada internacionalmente pela indústria de engenharia de software (GUEDES, 2018).

Neste projeto, a *UML* foi utilizada para a elaboração do diagrama de classes e diagrama de atividade.

3.2.5 Jest

O *Jest* [8] é um *framework* de testes em *JavaScript* desenvolvido para garantir a precisão e a confiabilidade do código. Ele fornece uma API intuitiva, rica em funcionalidades e fácil de usar, permitindo a criação de testes de maneira simples e eficiente. Com uma execução rápida, o *Jest* oferece feedback imediato, auxiliando os desenvolvedores na identificação e correção de erros no código *JavaScript* de forma ágil e eficaz.

Neste projeto, o *Jest* foi utilizado para garantir a qualidade e a funcionalidade da API, abrangendo tanto testes unitários quanto testes de integração. Isso assegurou que cada componente funcionasse corretamente de forma isolada e em conjunto com outros módulos, mantendo a estabilidade e a consistência da aplicação durante todo o processo de desenvolvimento.

3.2.6 Visual Studio Code

O *Visual Studio Code*[9] foi usado para escrever o código-fonte de todo o projeto. Ele oferece suporte a várias extensões e ferramentas de produtividade, como integração com o *Git* [10] e suporte a contêineres *Docker* [11], o que contribuiu para acelerar o desenvolvimento.

[8] Disponível em: https://jestjs.io/pt-BR/
[9] Disponível em https://code.visualstudio.com/
[10] Disponível em https://git-scm.com/
[11] Disponível em https://www.docker.com/

3.2.7 Postman

O *Postman*[12] foi utilizado para testar e documentar a API. Com o *Postman*, foi possível simular requisições HTTP e validar as respostas da API durante o desenvolvimento. A ferramenta também foi útil na automação de testes para garantir que a aplicação funcionasse conforme o esperado.

3.2.8 Docker

Docker [13] foi utilizado para criar contêineres que encapsulam o ambiente de desenvolvimento, garantindo que a *API* seja executada de forma consistente em diferentes plataformas. *Docker* também facilitou a implantação da aplicação no ambiente de produção.

3.3 Requisitos

Os requisitos funcionais correspondem às funcionalidades essenciais que o sistema precisa fornecer para atender às necessidades do problema proposto. Com base nas informações levantadas, foram identificadas as principais áreas do sistema. Cada requisito funcional é estruturado de forma a garantir a execução eficaz das operações e facilitar a interação entre as diversas partes do sistema. Os requisitos não funcionais definem critérios de qualidade e restrições do sistema.

A seguir, são listados os requisitos funcionais e os requisitos não funcionais em formato de história de usuários:

[12] Disponível em https://www.postman.com/
[13] Disponível em https://www.docker.com/

ID	História de Usuário	Descrição
RF01	Como usuário, quero cadastrar, atualizar, visualizar e deletar clientes na plataforma.	Permitir a gestão completa de clientes, incluindo nome, CPF, e-mail, telefone, endereço e status (ativo/inativo).
RF02	Como usuário, quero cadastrar, atualizar, visualizar e deletar ferramentas na plataforma.	Permitir a gestão completa de ferramentas, incluindo nome, descrição, valor da diária, imagem, status (ativo/inativo) e categoria da ferramenta.
RF03	Como usuário, quero cadastrar, atualizar, visualizar e deletar categorias de ferramentas na plataforma.	Permitir a gestão completa de categorias, incluindo nome, status e descrição.
RF04	Como usuário, quero cadastrar, atualizar, visualizar e deletar grupos na plataforma.	Permitir a gestão completa de grupos de usuários, incluindo nome, descrição, status (ativo/inativo) e permissões associadas.
RF05	Como usuário, quero cadastrar, atualizar, visualizar e deletar locações ativas.	Permitir a gestão completa das locações, associando clientes e ferramentas, incluindo data de início, data de fim, valores de descontos, multas e seguros.
RF06	Como usuário, quero visualizar uma lista de todas as locações ativas.	Exibir todas as locações em andamento, incluindo detalhes como cliente, ferramenta e data de devolução.
RF07	Como usuário, quero cadastrar, atualizar, visualizar e deletar revisões das ferramentas devolvidas.	Permitir a gestão completa das revisões, verificando o estado da ferramenta após devolução, incluindo nome da ferramenta e data da revisão.
RF08	Como usuário, quero cadastrar, atualizar, visualizar e deletar rotas com permissões e domínios.	Permitir a configuração de rotas no sistema, definindo permissões e domínios de acesso.
RF09	Como usuário, quero cadastrar, atualizar, visualizar e deletar usuários do sistema.	Permitir a gestão completa de usuários, incluindo nome, e-mail, senha, CPF, status (ativo/inativo) e os grupos aos quais pertencem.
RF10	Como usuário, quero acessar detalhes completos de uma ferramenta.	Exibir informações como histórico de locações e revisões realizadas.

Tabela 1 – Tabela de Requisitos Funcionais

ID	Requisito Não Funcional	Descrição
RNF01	O sistema deve ser desenvolvido utilizando JavaScript, Node.js, Express.js e MongoDB.	Definir a stack de tecnologia utilizada para o desenvolvimento da aplicação.
RNF02	A API deve ser estruturada seguindo boas práticas de organização de rotas e middlewares.	Garantir uma arquitetura bem organizada e modular para facilitar a manutenção e escalabilidade do código.
RNF03	O sistema deve garantir alta disponibilidade e suportar múltiplas requisições simultâneas.	O sistema deve ser capaz de lidar com grande volume de acessos sem interrupções no serviço.
RNF04	O sistema deve ser hospedado em um ambiente que suporte Docker.	Permitir a containerização da aplicação para facilitar o deployment e a escalabilidade.
RNF05	A interface de gerenciamento deve ser responsiva, adaptando-se a diferentes dispositivos.	Garantir que a interface funcione corretamente em telas de diferentes tamanhos e resoluções.
RNF06	O tempo de resposta das requisições da API não deve ultrapassar 2 segundos em condições normais.	Assegurar um desempenho adequado para evitar lentidão na experiência do usuário.
RNF07	O banco de dados deve permitir escalabilidade.	Suportar aumento no volume de dados sem degradação significativa de desempenho.
RNF08	A documentação da API deve ser disponibilizada via Swagger.	Facilitar o entendimento e a integração com a API por outros desenvolvedores.
RNF09	O sistema deve ser capaz de processar até 500 solicitações simultâneas.	Garantir estabilidade e eficiência mesmo sob alta demanda.
RNF10	O sistema deve ter uma disponibilidade mínima de 99,9% durante o horário comercial.	Evitar indisponibilidade do sistema durante períodos críticos de uso.
RNF11	O sistema deve garantir a segurança das informações sensíveis.	Implementar criptografia e controle de acesso para proteger dados como CPF, e-mails e senhas.
RNF12	O sistema deve realizar backups diários dos dados importantes.	Assegurar que dados possam ser restaurados dentro de 2 horas em caso de falha.

Tabela 2 – Tabela de Requisitos Não Funcionais

3.4 Arquitetura do Software

A estrutura escolhida para a API é a Arquitetura em Camadas, que organiza o sistema em diferentes níveis de responsabilidade, permitindo uma gestão mais eficaz e facilitando a manutenção e o avanço contínuo do sistema. Essa arquitetura é composta por várias camadas, que interagem de forma hierárquica para garantir o bom funcionamento do sistema.

As principais camadas em uma Arquitetura em Camadas são:

- Camada de Apresentação: Responsável por receber as requisições dos consumidores da API e retornar respostas estruturadas. Como a aplicação não possui uma interface gráfica, essa camada é composta pelos *endpoints* expostos, permitindo que usuários externos interajam com as funcionalidades do sistema por meio de requisições HTTP.
  - Interação: Ela se comunica diretamente com a Camada de Controle (*Controller*), que gerencia as requisições, valida os dados recebidos e direciona o fluxo para a Camada de Serviço.
  - Formato de Respostas: Os dados retornados pela API são estruturados no formato JSON, garantindo padronização e compatibilidade com diferentes aplicações clientes.
  - Ferramentas de Interação: O consumo da API pode ser realizado por meio de ferramentas como *Swagger* e *Postman*, que permitem testar e visualizar as respostas sem a necessidade de uma interface gráfica.
- Camada de Controle (Controller): Atua como intermediária entre as requisições externas e a lógica de negócio do sistema. Essa camada recebe as solicitações da API, valida os dados, processa as informações e encaminha os dados para a Camada de Serviço.
  - Interação: Recebe as requisições da Camada de Apresentação e direciona os dados para a Camada de Serviço.
- Camada de Serviço (ou Lógica de Negócio): Responsável por implementar as regras de negócio da aplicação. Aqui ocorrem as validações e processamentos necessários para garantir que as informações sejam tratadas conforme os requisitos do sistema.
  - Interação: Processa as informações recebidas do Controller e acessa a Camada de Persistência para buscar ou armazenar dados no banco de dados.
- Camada de Persistência (ou Repositório): Responsável pelo armazenamento e recuperação dos dados no banco de dados. Essa camada lida diretamente com a persistência das informações, garantindo sua integridade e disponibilidade.

- Interação: A Camada de Serviço consulta e modifica os dados armazenados no banco por meio do Repositório.

3.4.1 Fluxo de Interação entre as Camadas

A comunicação entre as camadas ocorre de maneira estruturada, garantindo a organização e o bom funcionamento do sistema. O fluxo segue a seguinte sequência:

1. A Camada de Apresentação recebe a requisição do consumidor da API e a encaminha ao *Controller*.

2. O *Controller* processa a solicitação e repassa os dados para a Camada de Serviço, onde as regras de negócio são aplicadas.

3. A Camada de Serviço consulta ou modifica os dados no banco de dados por meio da Camada de Persistência.

4. A Camada de Persistência retorna os dados processados à Camada de Serviço.

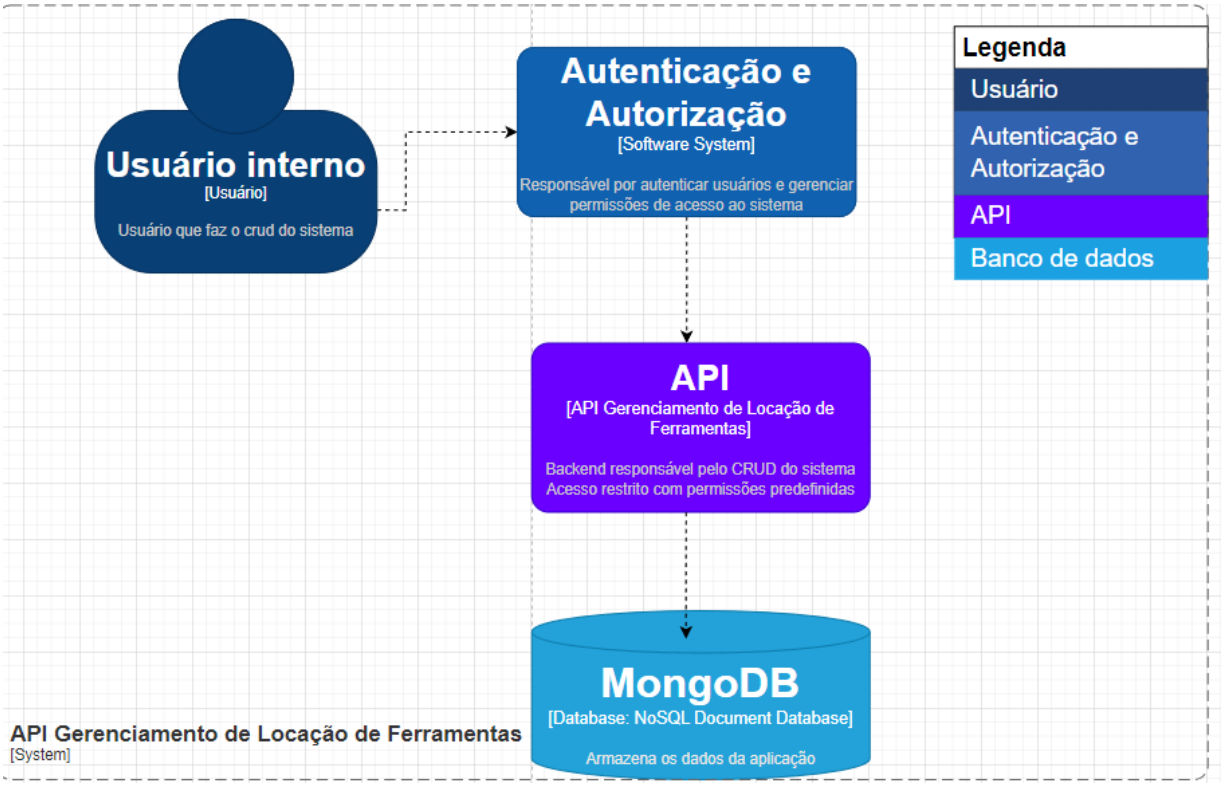
5. A Camada de Serviço envia os dados processados ao *Controller*, que os formata e os retorna à Camada de Apresentação.

6. A Camada de Apresentação envia a resposta final ao consumidor da API.

Essa separação de responsabilidades torna o sistema mais modular, permitindo que cada camada seja testada e aprimorada de forma independente. Além disso, essa estrutura facilita a manutenção e possibilita a evolução contínua da aplicação, garantindo maior organização e eficiência no desenvolvimento da API.

Na figura abaixo, é demonstrada a arquitetura do sistema baseada no modelo de Contexto do C4 *Model*:

Figura 5 – Arquitetura

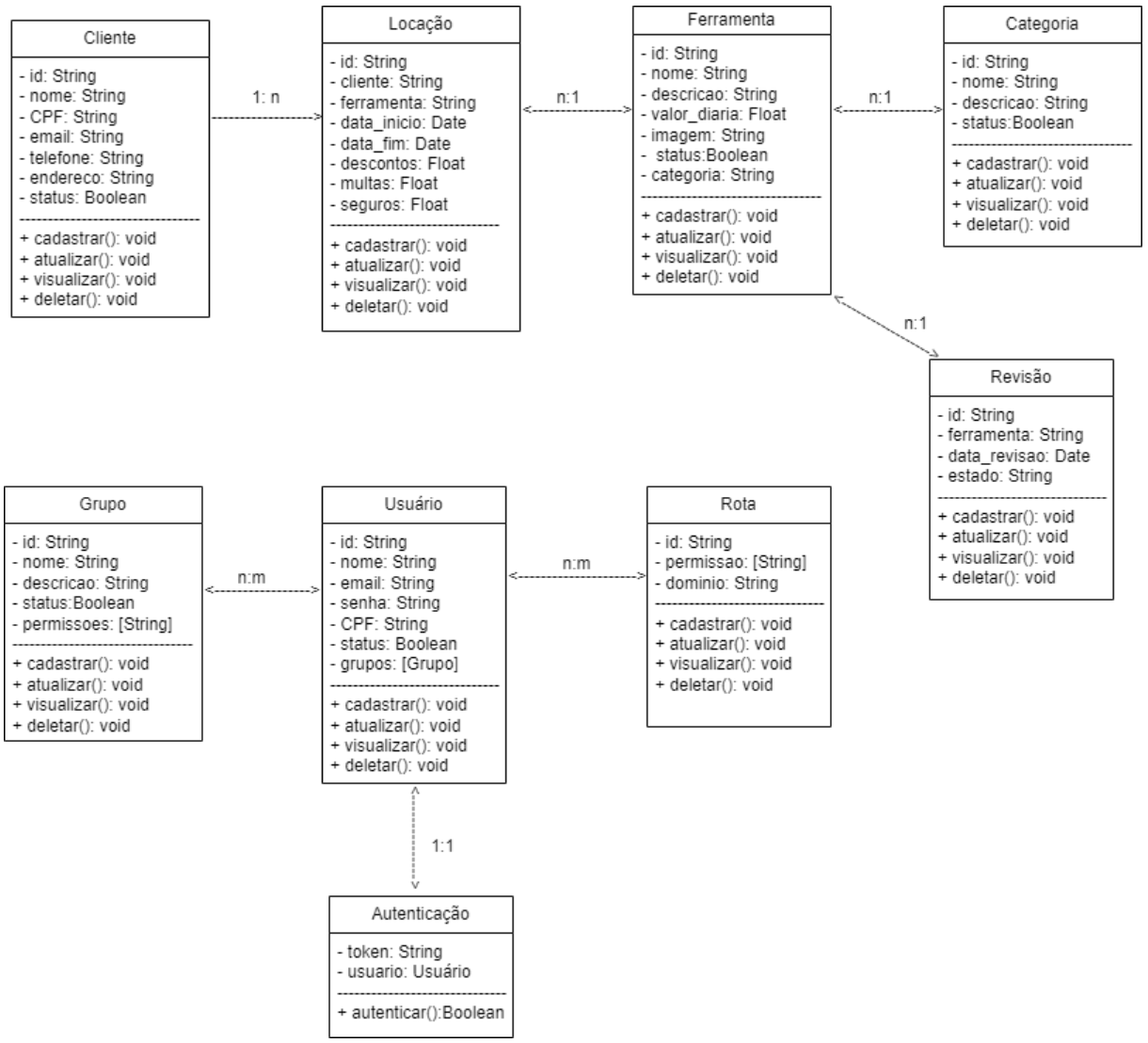


Fonte: elaborado pelo autor (2025).

3.5 Modelagem

Ao longo da documentação do projeto, foram criados dois diagramas *UML* - (*Unified Modeling Language*) (GUEDES, 2018). Dentre eles, o diagrama de classes tem o objetivo de ilustrar a estrutura e a modelagem das classes, evidenciando como os dados se inter-relacionam dentro da aplicação e o diagrama de atividades.

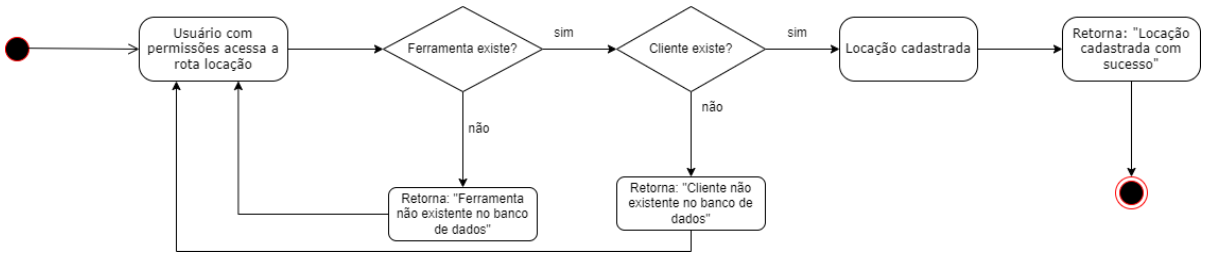
Figura 6 – Diagrama de classes



Fonte: elaborado pelo autor (2025).

A figura abaixo demonstra o diagrama de atividade; nele, é mostrado o fluxo de execução da API para o cadastro de uma locação.

Figura 7 – Diagrama de atividade



Fonte: elaborado pelo autor (2025).

3.6 Persistência de Dados

O banco de dados *NoSQL MongoDB* [14] foi utilizado para armazenar as informações do sistema, pois ele oferece flexibilidade. Para gerenciar a estrutura dos dados, empregou-se a biblioteca *Mongoose* [15] que permitiu definir esquemas e garantir a validação das informações cadastradas, como usuários, clientes, ferramentas, categorias, locações, grupos, revisões e rotas, além das rotas para autenticação e recuperação de senha.

A aplicação possui o total de dez rotas para manipulação dos dados e autenticação, mas, para manter a objetividade, serão apresentadas apenas as principais. Essas rotas são responsáveis pelo gerenciamento de locações de ferramentas, assegurando o funcionamento eficiente do sistema e o armazenamento correto dos dados. Entre elas, rota de locação é importante para o gerenciamento do sistema, pois permite registrar e controlar as locações de ferramentas para os clientes. Cada locação armazena informações, como a ferramenta alugada, o cliente que realizou a locação, as datas de início e fim, além dos valores envolvidos, como descontos, multas e seguro.

Para garantir que o valor total da locação seja sempre positivo, foi implementada uma validação específica. O *Mongoose* foi utilizado para estruturar os dados e o *mongoose-paginate* para facilitar a paginação dos registros, melhorando a eficiência das consultas no banco de dados.

[14] Disponível em: https://www.mongodb.com/
[15] Disponível em: Disponível em: https://www.mongodb.com/developer/languages/javascript/getting-started-with-mongodb-and-mongoose/

Figura 8 – *Schema* Locação

```
import mongoose from 'mongoose';
import mongoosePaginate from 'mongoose-paginate-v2';

const LocacaoSchema = new mongoose.Schema({
  ferramenta: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'ferramentas',
    required: true
  },
  cliente: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'clientes',
    required: true
  },
  inicio: {
    type: Date,
    required: true
  },
  fim: {
    type: Date,
    required: true
  },
  valor_desconto: {
    type: Number,
    required: true
  },
  valor_multa: {
    type: Number,
    required: true
  },
  valor_seguro: {
    type: Number,
    required: true
  },
  // (valor_diaria * dias de locacao) - descontos + multas + seguro
  valor_total: {
    type: Number,
    required: true,
    validate: {
      validator: function (v) {
        return v >= 0;
      },
      message: props => `${props.value} é um valor negativo.`
    }
  },
```

Fonte: elaborado pelo autor (2025).

A rota de ferramenta é essencial para o gerenciamento das ferramentas disponíveis

para locação. Ela armazena informações como o nome, descrição, valor diário e a imagem que pode representar a ferramenta, facilitando a identificação dos itens. Além disso, cada ferramenta está associada a uma categoria, que ajuda a organizá-las de forma eficiente no sistema. Já o campo ativo define se a ferramenta está disponível para locação ou não, e o valor diário reflete o custo de locação por dia.

Figura 9 – *Schema* Ferramenta

api-gerenciamento > src > models > Ferramenta.js > default

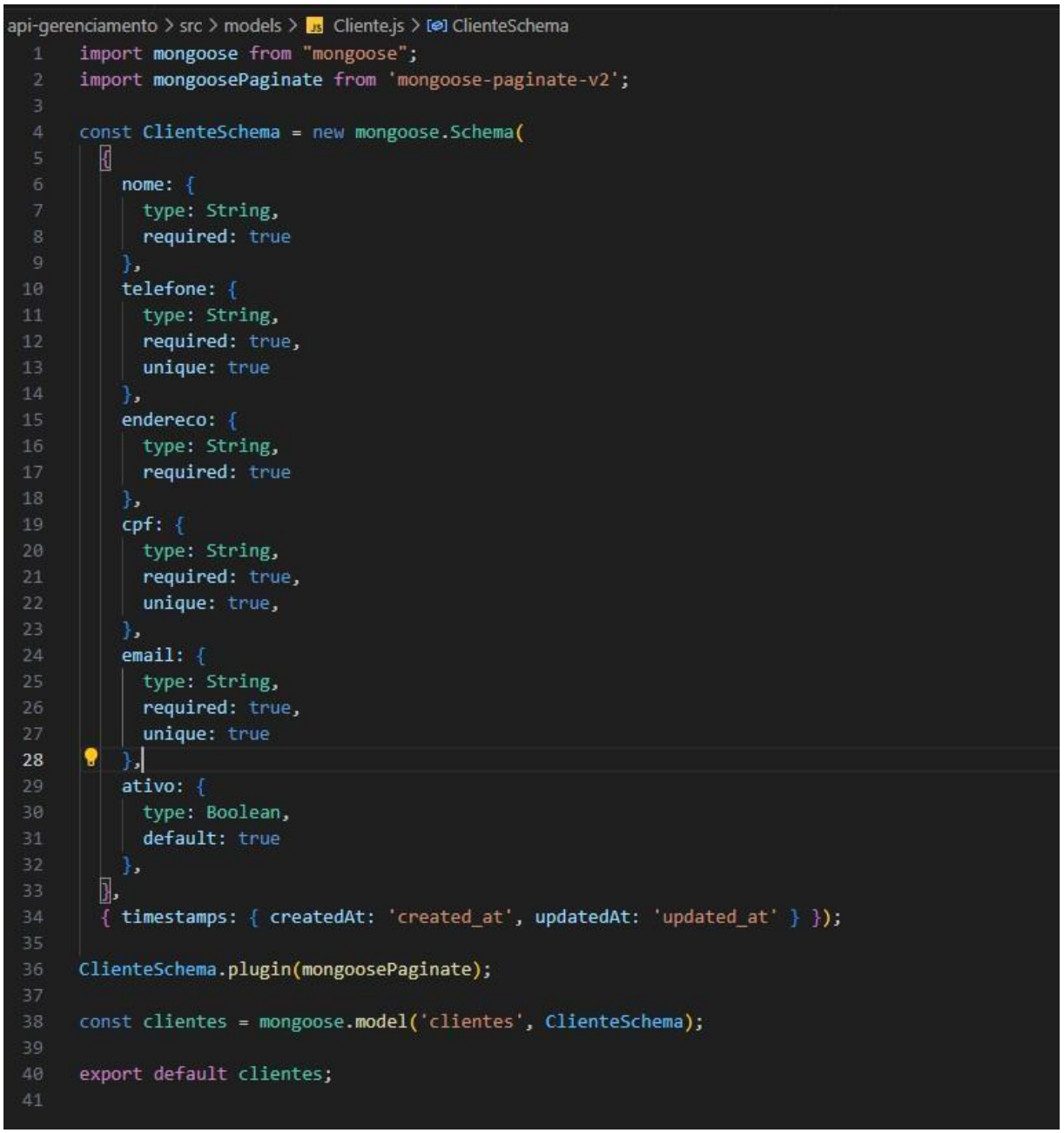
```
import mongoose from 'mongoose';
import mongoosePaginate from 'mongoose-paginate-v2';

const FerramentaSchema = new mongoose.Schema({
  nome: {
    type: String,
    unique: true,
    required: true
  },
  descricao: {
    type: String,

  },
  valor_diaria: {
    type: Number,
    required: true
  },
  imagem_path: {
    type: String
  },
  ativo: {
    type: Boolean,
    default: true
  },
  categoria: {
    required: true,
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Categoria'
  }
},
  { timestamps: { createdAt: 'created_at', updatedAt: 'updated_at' } });

FerramentaSchema.plugin(mongoosePaginate);

const ferramentas = mongoose.model('ferramentas', FerramentaSchema);

export default ferramentas;
```

Fonte: elaborado pelo autor (2025).

A rota de cliente é fundamental para o cadastro e gerenciamento das informações

dos clientes no sistema. Cada cliente tem dados pessoais, como nome, telefone, endereço, CPF e e-mail. Além disso, o campo "ativo"indica se o cliente está ativo no sistema.

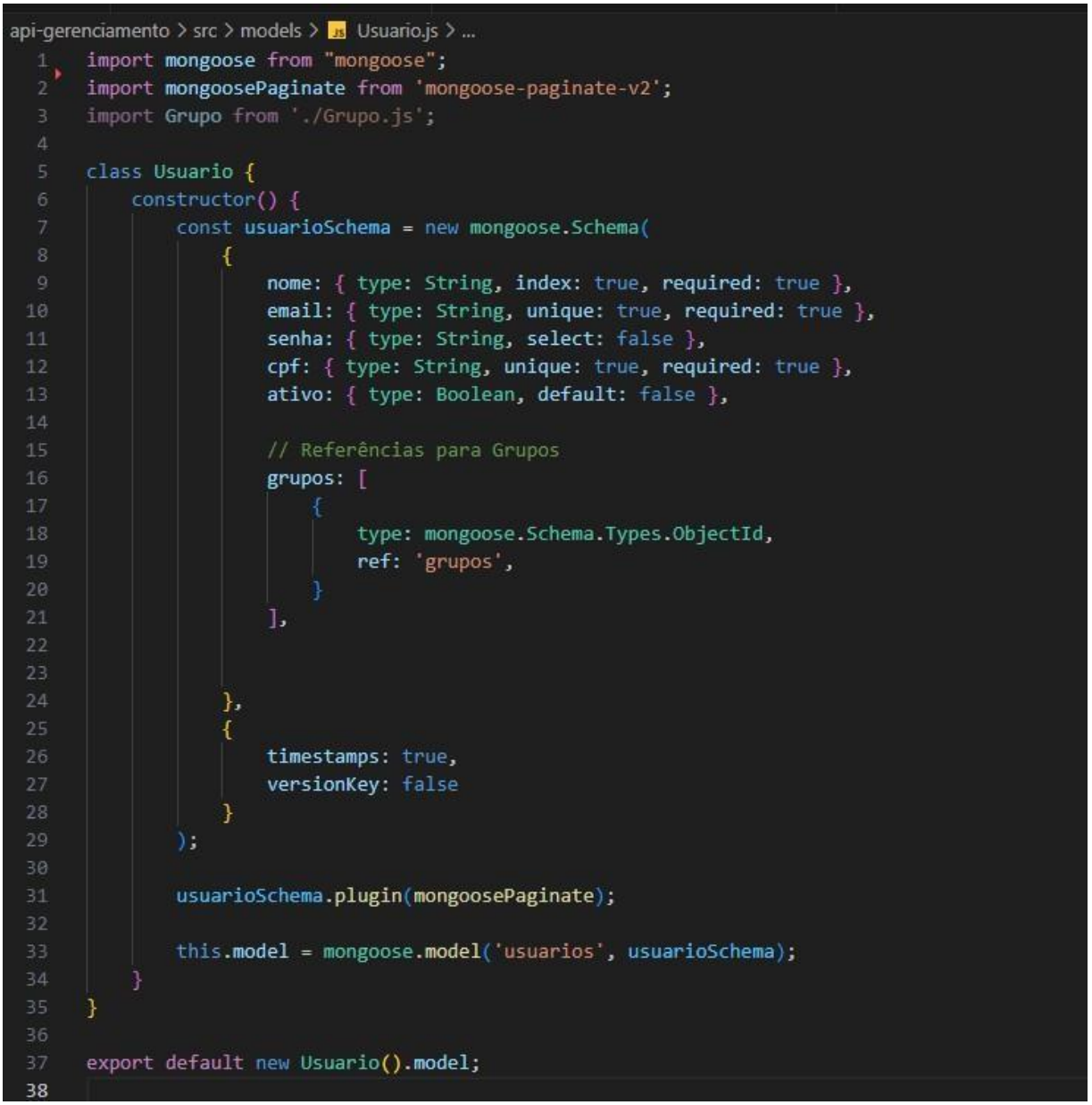
Figura 10 – *Schema* Cliente



Fonte: elaborado pelo autor (2025).

A rota de usuário é responsável pelo gerenciamento completo dos usuários, permitindo o cadastro, consulta e controle de acessos. Cada usuário possui informações essenciais, como nome, e-mail, senha e CPF, além de um status que indica se está ativo no sistema.

Figura 11 – *Schema* Usuário



Fonte: elaborado pelo autor (2025).

3.7 Licença de Uso

A licença utilizada no desenvolvimento desse projeto foi a licença "*MIT*"[16] . Uma licença para software livre desenvolvida pelo Instituto de Tecnologia de Massachusetts.

[16] Disponível em: https://opensource.org/license/MIT

4 Resultados e discussões

4.1 Gerenciamento de configuração e mudanças

Para o gerenciamento do código-fonte e controle de versão do projeto, foi escolhida a plataforma *GitLab* [1], que proporcionou uma organização eficiente do código e facilitou o acompanhamento das alterações realizadas ao longo do desenvolvimento. De acordo com (LOELIGER; MCCULLOUGH, 2012), o uso de sistemas de controle de versão como o *Git* possibilita um fluxo de trabalho organizado, permitindo rastreamento eficiente das mudanças e colaboração entre desenvolvedores de maneira segura e controlada.

O repositório do projeto está disponível no *GitLab* e pode ser acessado através do seguinte link: https://gitlab.fslab.dev/AminahMakhoul1/api-gerenciamento

No *GitLab*, foram configuradas três branches principais, cada uma com um objetivo específico:

- *development*: *branch* dedicada ao desenvolvimento contínuo, onde novas funcionalidades e correções eram implementadas e testadas. O código era integrado após revisão e *merge*.

- *master* : representa a versão estável e pronta para produção, contendo apenas código validado e testado, utilizado para gerar *releases*.

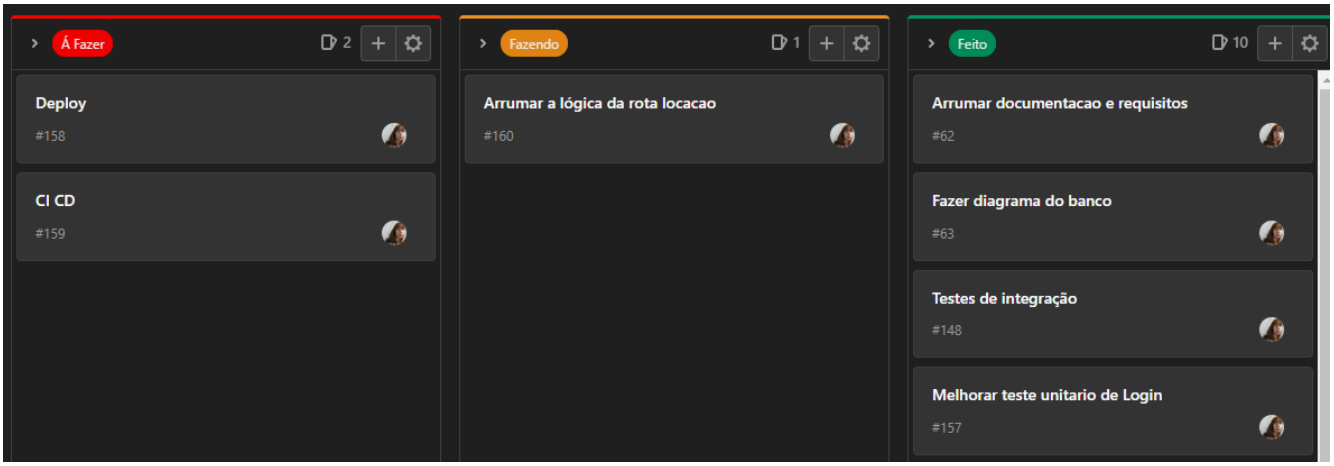
- testar-ci-cd: utilizada para testar as configurações de integração e entrega contínua, garantindo que as *pipelines* funcionassem corretamente.

4.2 Processo de desenvolvimento

A metodologia *Kanban* foi escolhida para o desenvolvimento do projeto, já que é muito flexível e fácil de usar. Uma das grandes vantagens do *Kanban* é que ele permite que as tarefas sejam executadas conforme a prioridade, sem a necessidade de seguir uma ordem específica. Isso deu mais liberdade para organizar as tarefas e adaptá-las conforme surgiam demandas do projeto. Conforme ilustrado na figura 12, é apresentado o quadro *Kanban* utilizado ao longo do desenvolvimento do projeto.

[1] Disponível em : https://about.gitlab.com/

Figura 12 – Quadro *Kanban*



Fonte: elaborado pelo autor (2025).

- A fazer: Nesta coluna estão as tarefas que ainda não foram iniciadas, mas que já foram planejadas e aguardam para serem começadas.

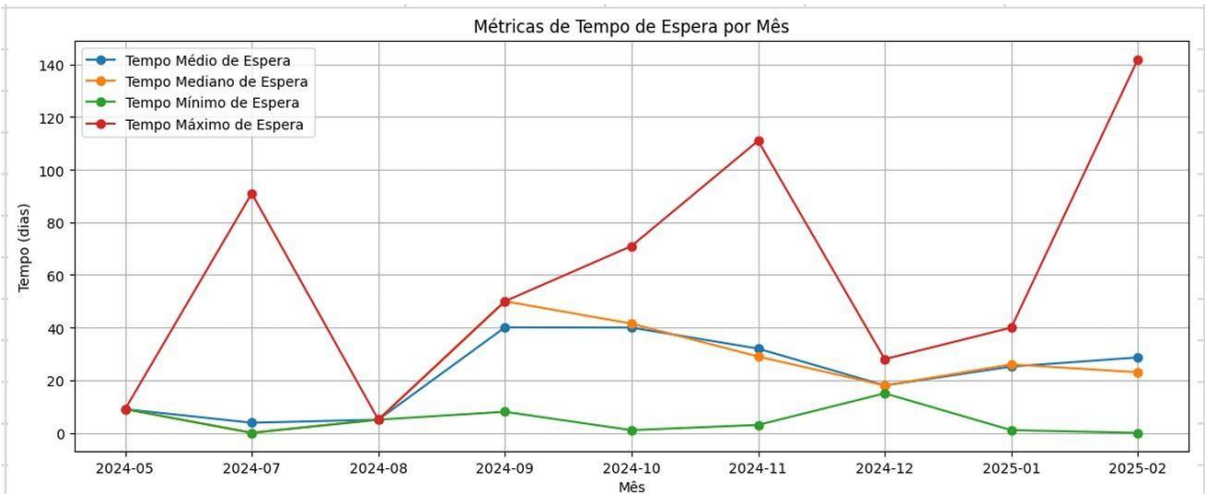
- Fazendo: Aqui ficam as tarefas que estão em execução no momento. Elas são movidas para esta coluna assim que o trabalho sobre elas é iniciado.

- Feito: Nesta coluna estão as tarefas que já foram concluídas, ou seja, que passaram por todas as etapas necessárias e foram finalizadas.

4.3 Gerenciamento de tarefas - Métricas

Para entender melhor o desenvolvimento deste trabalho, foram gerados gráficos que apresentam indicadores importantes: *Lead Time* (Tempo Total de Processamento), *Cycle Time* (Tempo de Execução), *WIP - Work in Progress* (Trabalho em Andamento) e *Throughput* (Taxa de Conclusão). Esses indicadores permitem visualizar o tempo necessário para a conclusão das tarefas, a quantidade de trabalho em andamento e a eficiência na finalização das demandas. A seguir, cada métrica será explicada e analisada com base nos gráficos gerados.

Figura 13 – *Lead Time*



Fonte: elaborado pelo autor (2025).

O gráfico da figura 13 mostra o *Lead Time*, ou seja, o tempo total desde a solicitação até a conclusão de cada tarefa. Esse indicador é essencial para entender quanto tempo, em média, uma tarefa ficou em aberto antes de ser finalizada. Ao analisar o gráfico, percebe-se que esse tempo variou bastante ao longo dos meses:

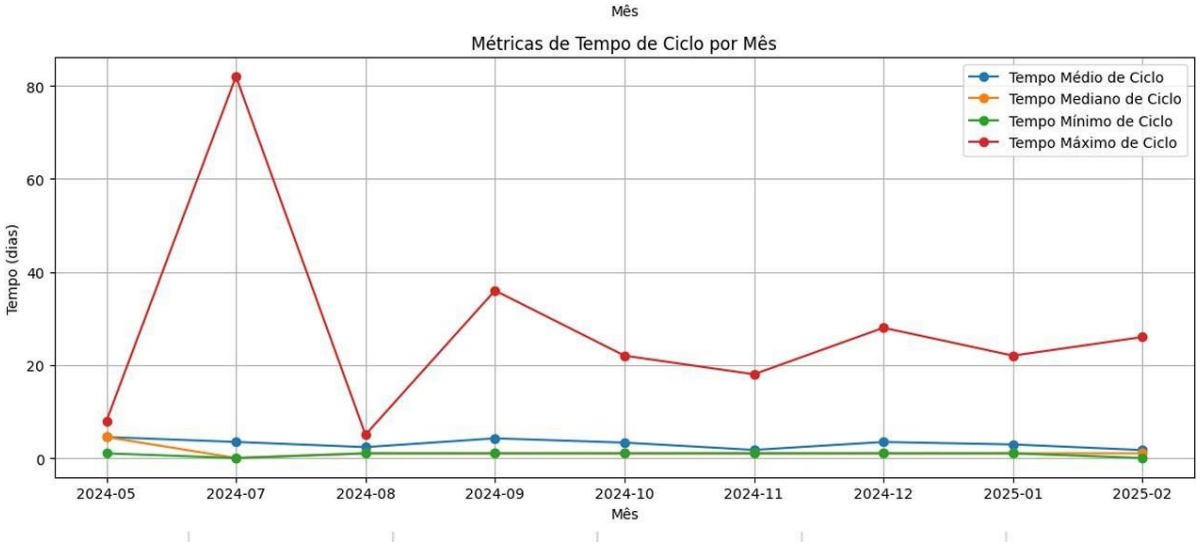
- Julho de 2024: A maioria das tarefas foi resolvida no mesmo dia em que foi criada, o que fez com que a mediana fosse 0 dias. No entanto, houve um caso em que uma demanda levou 91 dias para ser finalizada, o que elevou a média para 3,83 dias.

- Setembro de 2024: O tempo médio de espera aumentou, chegando a 40,08 dias, com metade das tarefas levando 50 dias ou mais para serem concluídas. Isso mostra que algumas demandas ficaram paradas por um longo período antes de serem resolvidas.

- Fevereiro de 2025: Esse foi o mês com o maior tempo médio registrado: 28,65 dias. Além disso, uma das tarefas levou 142 dias para ser finalizada, o que indica que, em certos casos, o tempo de conclusão das atividades foi muito maior do que o esperado.

Essas variações mostram que, em alguns momentos, o fluxo de tarefas funcionou de maneira eficiente, com demandas sendo concluídas rapidamente. No entanto, há períodos em que as tarefas se acumularam e levaram muito mais tempo para serem finalizadas. Monitorar esse indicador ajuda a identificar atrasos e otimizar o fluxo de trabalho para maior agilidade e eficiência.

Figura 14 – *Cycle Time*



Fonte: elaborado pelo autor (2025).

O gráfico da figura 14 representa o *Cycle Time*, ou seja, o tempo total desde a solicitação até a conclusão de cada tarefa. Diferente do *Lead Time*, que considera todo o tempo da demanda no sistema, ele foca apenas no tempo real de execução. No geral, a maioria das tarefas foi concluída rapidamente, com a mediana girando em torno de 1 dia. No entanto, alguns meses apresentaram exceções:

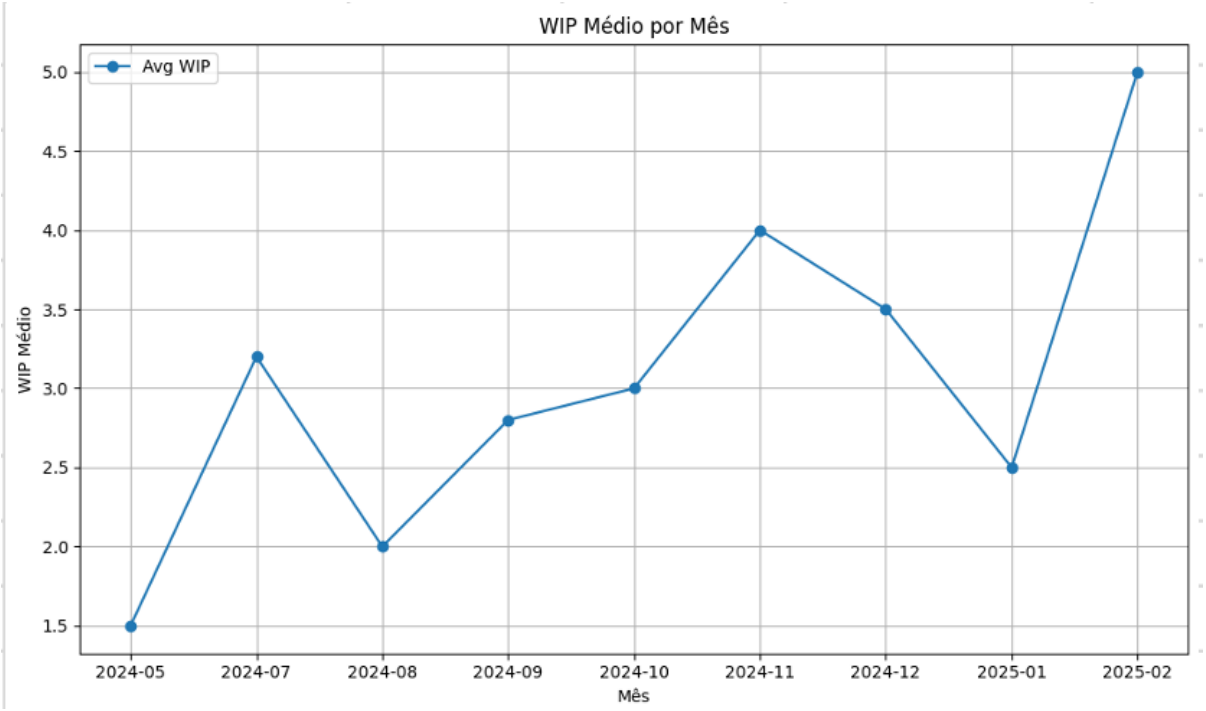
- Setembro de 2024: Algumas tarefas levaram até 36 dias para serem concluídas, devido a mudanças de prioridades ao longo do processo.

- Outubro de 2024: O tempo médio aumentou para 3,31 dias, com um máximo de 22 dias, mostrando um pequeno aumento no tempo de execução.

- Fevereiro de 2025: Apesar do alto *Lead Time*, o *Cycle Time* médio foi de apenas 1,70 dias, indicando que a principal demora ocorreu antes do início da execução da tarefa, e não durante sua realização.

Esses dados mostram que, uma vez iniciadas, as tarefas geralmente eram concluídas rapidamente.

Figura 15 – *Work in Progress*



Fonte: elaborado pelo autor (2025).

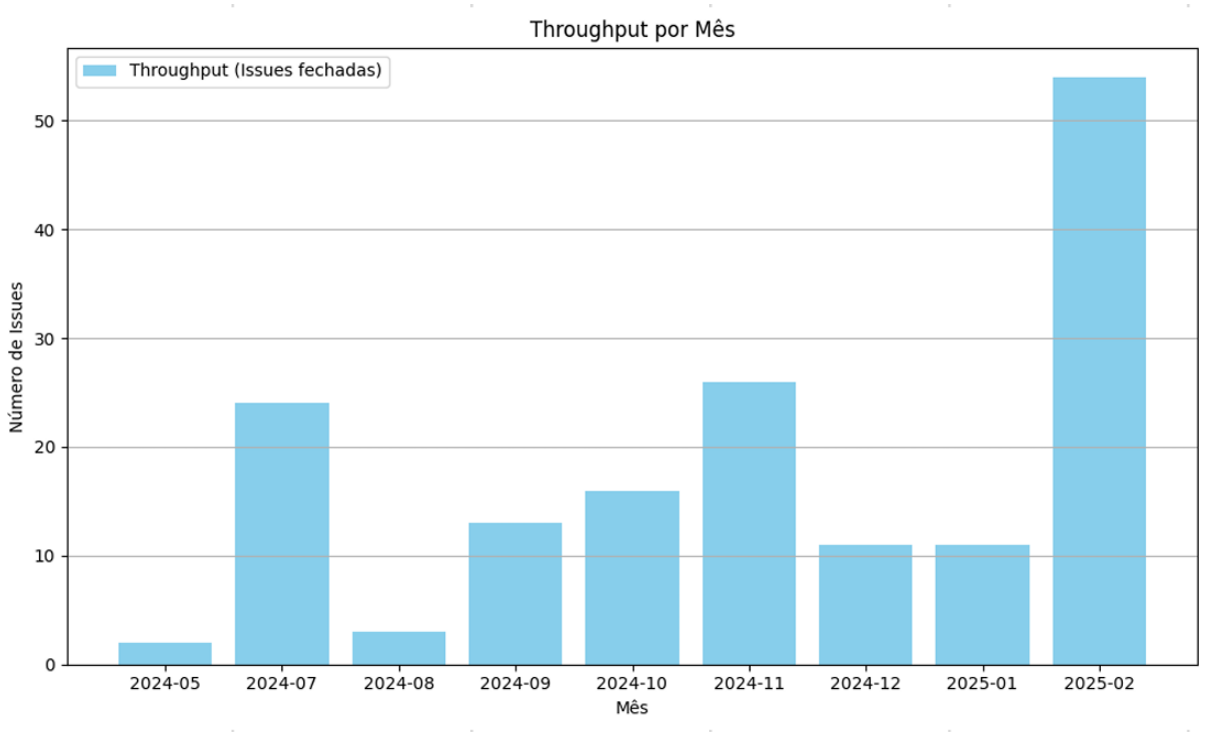
O *WIP (Work in Progress)*, mostrado na figura 15, representa quantas tarefas ficaram em andamento ao longo do tempo, ou seja, quantas demandas foram abertas e ainda não tinham sido finalizadas. Esse indicador ajuda a entender como o fluxo de trabalho se comportou ao longo dos meses. O gráfico mostra que o *WIP* foi aumentando com o tempo, ou seja, mais tarefas ficaram abertas antes de serem concluídas:

- Maio de 2024: O *WIP* médio foi de 1,5 tarefas, indicando que poucas demandas estavam em aberto.

- Fevereiro de 2025: O *WIP* chegou ao maior valor registrado, com 5 tarefas em andamento, mostrando que mais demandas estavam acumuladas ao mesmo tempo.

Se o *WIP* cresce demais, significa que muitas tarefas foram iniciadas, mas poucas foram finalizadas no mesmo ritmo, o que pode acabar causando atrasos. Acompanhar esse indicador é essencial para evitar acúmulos e garantir que o fluxo de trabalho siga de forma mais equilibrada e eficiente.

Figura 16 – *Throughput*



Fonte: elaborado pelo autor (2025).

O gráfico de *Throughput*, apresentado na figura 16, mostra a quantidade de atividades concluídas por mês. Observa-se que houve variações significativas ao longo do tempo, refletindo mudanças no ritmo de execução das tarefas.

- Maio de 2024: Foi um dos meses de menor produtividade, com apenas 2 tarefas concluídas. Esse baixo volume de atividades está relacionado ao fato de que a API ainda não havia sido desenvolvida, resultando em poucas demandas a serem finalizadas.
- Julho de 2024: Registrou um aumento expressivo, com 24 tarefas concluídas, indicando um crescimento na capacidade de entrega.
- Fevereiro de 2025: Apresentou o maior pico de produtividade, com 54 tarefas finalizadas, sinalizando um fluxo de trabalho mais eficiente.

O aumento do *Throughput* mostra que mais tarefas foram concluídas, o que é um bom sinal. Porém, é importante que esse crescimento aconteça de forma equilibrada, garantindo que as entregas mantenham sua qualidade e que o fluxo de trabalho não fique sobrecarregado.

4.4 Plano de testes

Nesta seção, serão descritos os testes realizados durante o processo de desenvolvimento da API, com o objetivo de garantir a qualidade e a confiabilidade do código. Os testes foram fundamentais para validar o funcionamento de cada funcionalidade implementada, desde os testes unitários até os testes de integração.

4.4.1 Introdução

4.4.1.1 Objetivo

Garantir a qualidade e a confiabilidade da API, validando suas funcionalidades, a integridade dos dados e a comunicação com o usuário.

4.4.1.2 Escopo

Os testes abrangem:

- Validação dos *endpoints GET, POST, PUT, DELETE* em diferentes cenários.
- Testes unitários de funções e métodos individuais.
- Testes de integração para verificar a comunicação entre módulos e banco de dados.
- Testes manuais e automatizados para identificar falhas e validar regras de negócio.

4.4.2 Estratégia de testes

4.4.2.1 Abordagem

- Testes Manuais: Utilização do *Postman* [2] para simulação de requisições HTTP e validação das principais rotas.
- Testes Automatizados: Implementação com *Jest* [3] para garantir cobertura de código e *feedback* imediato.

4.4.3 Tipos de Testes

- Testes Unitários: Verificam funções e métodos isoladamente.
- Testes de Integração: Avaliam a comunicação entre os módulos da API e o banco de dados.

[2] Disponivel em: https://www.postman.com/
[3] Disponivel em: https://jestjs.io/pt-BR/

4.4.4 Ambiente de Testes

4.4.4.1 Configuração do Ambiente

- Os testes foram realizados no próprio ambiente de desenvolvimento, utilizando o *VS Code*.
- Foi criada uma pasta específica dentro do projeto para organizar os testes.
- Para evitar impactos nos dados de produção, foi criado um banco de dados paralelo exclusivo para o ambiente de testes, garantindo maior segurança nos resultados.

4.4.5 Ferramentas Utilizadas

- *Postman* [4]: Para testes manuais de requisições HTTP.
- *Jest* [5]: Para execução dos testes automatizados.

4.4.6 Casos de Teste

4.4.6.1 Testes Manuais

Objetivo: Verificar a resposta da API e validar as regras de negócio antes da implementação dos testes automatizados.

Metodologia:

- Simulação de requisições HTTP no *Postman* para avaliar as respostas da API.
- Identificação e correção de erros iniciais, como falhas de validação e retornos inesperados.

4.4.6.2 Testes Automatizados

Ferramenta Utilizada: *Jest*, escolhida por sua simplicidade, rápida execução e suporte a mocks.

Categorias de Testes:

- Testes Unitários: Avaliação de funções e métodos isoladamente.
- Testes de Integração: Verificação da comunicação entre módulos e banco de dados.

[4] Disponível em: https://www.postman.com/
[5] Disponível em: https://jestjs.io/pt-BR/

4.4.7 Casos de Teste por Tipo de *Endpoint*

- *GET Endpoints:*
  - Cenário Positivo: Retorna status 200 com os dados corretos.
  - Cenário Negativo: Retorna 404 para um ID inexistente.
- *POST Endpoints*:
  - Cenário Positivo: Criação bem-sucedida de um recurso com status 201.
  - Cenário Negativo: Retorno 400 para requisições com dados inválidos.
- *PUT Endpoints*:
  - Cenário Positivo: Atualização de um recurso existente com status 200.
  - Cenário Negativo: Retorno 404 ao tentar atualizar um recurso inexistente.
- *DELETE Endpoints*:
  - Cenário Positivo: Exclusão bem-sucedida de um recurso com status 200.
  - Cenário Negativo: Retorno 404 para exclusão de um recurso que não existe.

4.5 Relatório dos Testes

4.5.1 Resultados dos Testes Manuais

- Confirmação das respostas esperadas nos *endpoints*.
- Validação das regras de negócio implementadas.
- Identificação e correção de falhas antes da automação dos testes.

4.5.2 Resultados dos Testes Automatizados

- Testes unitários confirmaram que funções e métodos operam corretamente de forma isolada.
- Testes de integração validaram a comunicação entre os módulos da API.
- *Feedback* imediato permitiu correção rápida de erros.

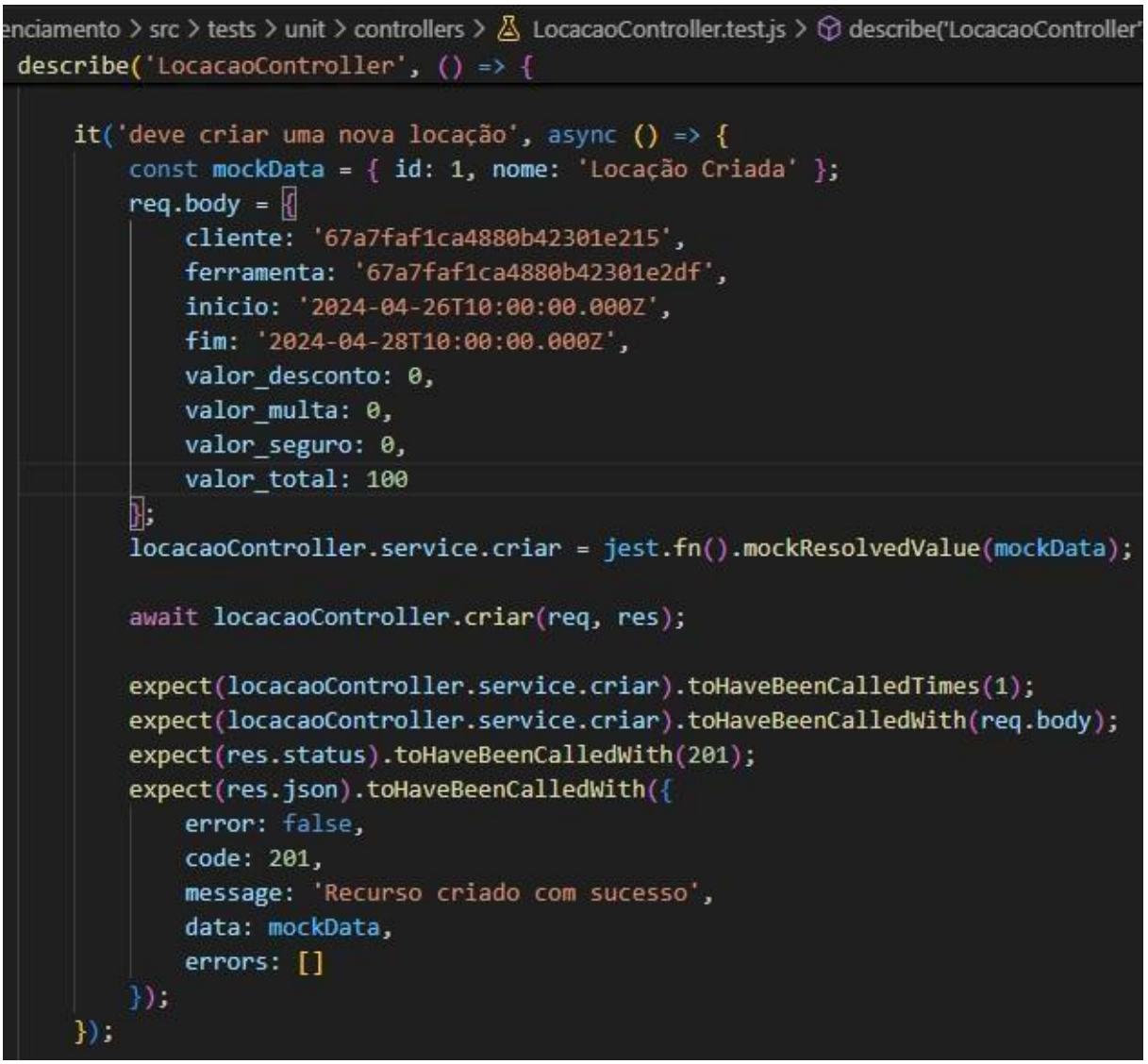
4.6 Conclusão

Os testes realizados garantiram a confiabilidade da API, validando tanto suas funcionalidades individuais quanto sua integração com o banco de dados. A execução de testes automatizados com *Jest* permitiu maior cobertura e agilidade na detecção de falhas, assegurando a qualidade da API antes do *deploy*.

4.7 Imagens e Resultados dos Testes

A seguir, são apresentadas imagens que ilustram os resultados dos testes realizados. Elas exibem os relatórios de execução dos testes unitários e de integração, oferecendo uma visão detalhada da cobertura de código e do comportamento da API durante o processo de validação.

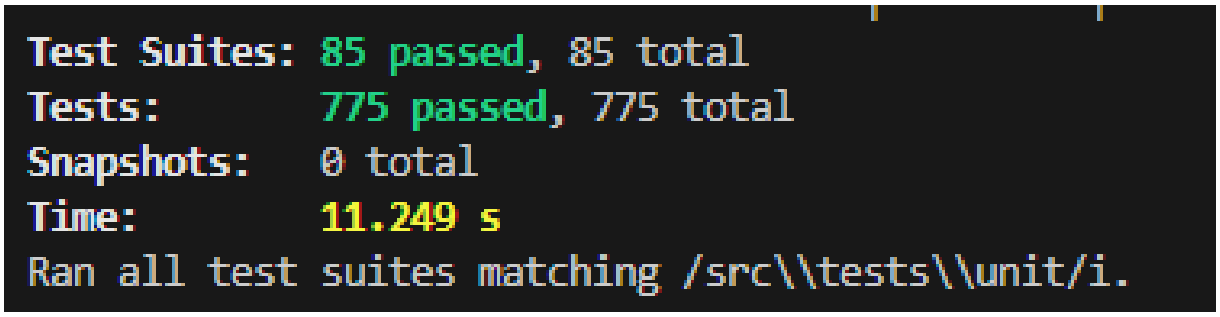
Figura 17 – Teste unitário Locação



Fonte: elaborado pelo autor (2025).

Na figura 17 é exibido um exemplo de teste unitário para a função de criação de locação. Esse teste verifica se uma locação é criada corretamente. Utilizando mocks, foi possível simular o retorno dos métodos de busca e garantir que o teste se concentrasse apenas na lógica interna da função. Isso assegura que a funcionalidade de criação de locação esteja funcionando conforme o esperado.

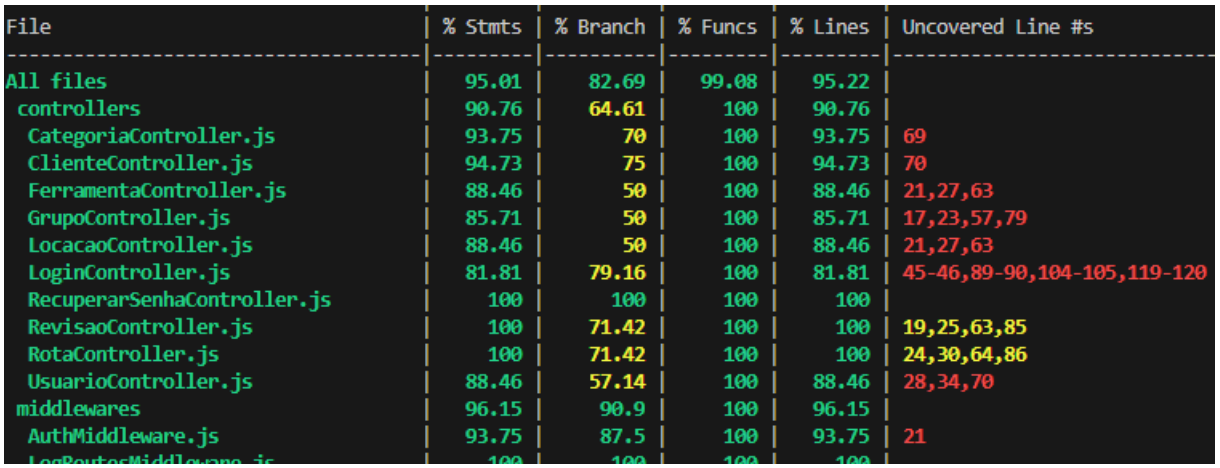
Figura 18 – Execução de testes unitários



Fonte: elaborado pelo autor (2025).

Na figura 18 é possível observar o resultado da execução dos testes unitários. Todas as 85 suítes de teste passaram com sucesso, totalizando 775 testes concluídos em aproximadamente 11.249 segundos. Isso demonstra a estabilidade e confiabilidade das funcionalidades implementadas.

Figura 19 – Cobertura de testes unitários



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	95.01	82.69	99.08	95.22	
controllers	90.76	64.61	100	90.76	
CategoriaController.js	93.75	70	100	93.75	69
ClienteController.js	94.73	75	100	94.73	70
FerramentaController.js	88.46	50	100	88.46	21,27,63
GrupoController.js	85.71	50	100	85.71	17,23,57,79
LocacaoController.js	88.46	50	100	88.46	21,27,63
LoginController.js	81.81	79.16	100	81.81	45-46,89-90,104-105,119-120
RecuperarSenhaController.js	100	100	100	100	
RevisaoController.js	100	71.42	100	100	19,25,63,85
RotaController.js	100	71.42	100	100	24,30,64,86
UsuarioController.js	88.46	57.14	100	88.46	28,34,70
middlewares	96.15	90.9	100	96.15	
AuthMiddleware.js	93.75	87.5	100	93.75	21

Fonte: elaborado pelo autor (2025).

A figura 19 apresenta o relatório de cobertura de testes unitários, detalhando o percentual de declarações executadas (%Stmts), ramificações condicionais (%Branch), funções testadas (%Funcs) e linhas de código cobertas (%Lines). A alta cobertura indica que

a maior parte do código foi testada, reduzindo a probabilidade de falhas não identificadas. Ao todo, a cobertura total dos testes unitários foi de 95.01%.

Figura 20 – Teste integração Locação

```
describe("Testes de integração - Locações", () => {
  it("Deve criar uma nova locação", async () => {
    const categoria = await Categoria.create({ nome: "Ferramentas", ativo: true });
    const cliente = await Cliente.create({
      nome: `Cliente ${Date.now()}`,
      telefone: `1199${Math.floor(100000 + Math.random() * 900000)}`,
      endereco: "Rua A, 123",
      cpf: `${Math.floor(10000000000 + Math.random() * 90000000000)}`,
      email: `cliente${Date.now()}@email.com`,
      ativo: true,
    });
    const ferramenta = await Ferramenta.create({
      nome: `Furadeira ${Date.now()}`,
      descricao: "Furadeira de impacto",
      ativo: true,
      valor_diaria: 10,
      imagem_path: "https://meusite.com/imagens/furadeira.jpg",
      categoria: categoria._id,
    });

    const locacaoData = {
      ferramenta: ferramenta._id,
      cliente: cliente._id,
      inicio: new Date("2025-03-10"),
      fim: new Date("2025-03-15"),
      valor_diaria: 10,
      valor_desconto: 0,
      valor_multa: 0,
      valor_seguro: 0,
      valor_total: 50,
    };

    const response = await request(app)
      .post("/locacoes")
      .set("Authorization", `Bearer ${authToken}`)
      .send(locacaoData)
      .expect(201);

    expect(response.body.data).toHaveProperty("_id");
  });
```

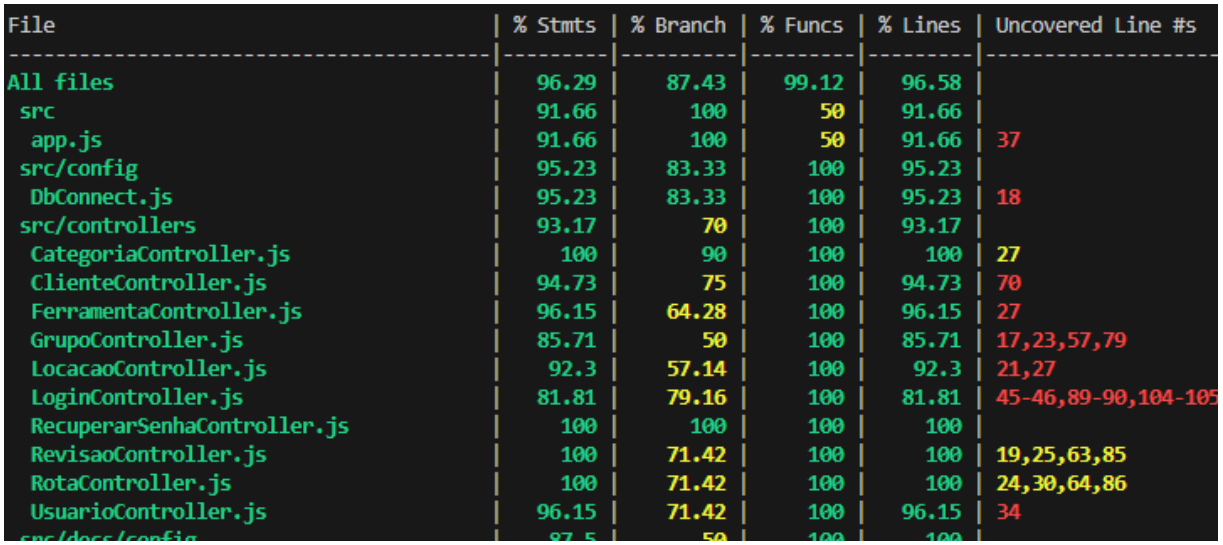
Fonte: elaborado pelo autor (2025).

O teste mostrado na figura 20 faz uma requisição *POST* para o *endpoint* que cadastra locações. Para que isso funcione, primeiro é criada uma categoria; logo, um cliente

e uma ferramenta, garantindo que todas as informações necessárias estejam disponíveis. Os dados são gerados dinamicamente para evitar qualquer conflito com registros no banco de dados. Depois, a locação é registrada com detalhes como período de uso, valores financeiros e a associação entre o cliente e a ferramenta escolhida.

A requisição é autenticada com um *token* de acesso e, ao receber a resposta da API, o teste valida se o código de *status* retornado é 201 *(Created)* e se a resposta contém um identificador único *(id)*. Essa abordagem assegura que a API está armazenando corretamente as locações e retornando os dados esperados.

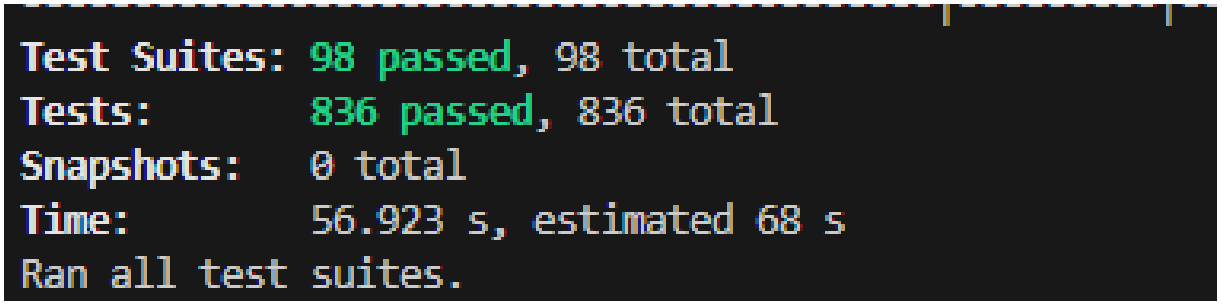
Figura 21 – Cobertura de todos os testes



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	96.29	87.43	99.12	96.58	
src	91.66	100	50	91.66	
app.js	91.66	100	50	91.66	37
src/config	95.23	83.33	100	95.23	
DbConnect.js	95.23	83.33	100	95.23	18
src/controllers	93.17	70	100	93.17	
CategoriaController.js	100	90	100	100	27
ClienteController.js	94.73	75	100	94.73	70
FerramentaController.js	96.15	64.28	100	96.15	27
GrupoController.js	85.71	50	100	85.71	17,23,57,79
LocacaoController.js	92.3	57.14	100	92.3	21,27
LoginController.js	81.81	79.16	100	81.81	45-46,89-90,104-105
RecuperarSenhaController.js	100	100	100	100	
RevisaoController.js	100	71.42	100	100	19,25,63,85
RotaController.js	100	71.42	100	100	24,30,64,86
UsuarioController.js	96.15	71.42	100	96.15	34

Fonte: elaborado pelo autor (2025).

Figura 22 – Quantidade de testes



Fonte: elaborado pelo autor (2025).

Nas figuras 21 e 22, é possível observar que a cobertura de testes, incluindo testes unitários e de integração, atingiu 96,29%, demonstrando a qualidade e a estabilidade da

aplicação. No total, foram realizados 836 testes, abrangendo diversos cenários e garantindo que as principais funcionalidades do sistema estejam funcionando corretamente.

4.8 Documentação

A documentação do projeto está dividida em dois componentes principais. O primeiro componente está no *Readme* do repositório, onde o desenvolvedor pode fazer o download do código-fonte e acessar as instruções para configurar o ambiente de desenvolvimento, conforme mostrado na figura 23. O segundo componente foi gerado utilizando o *Swagger* [6], uma ferramenta que oferece uma interface interativa e visual, permitindo aos desenvolvedores explorar e testar os *endpoints* de forma eficiente. Para facilitar o acesso a essa documentação, foi utilizada a biblioteca *Swagger UI Express*[7], que exibe a documentação gerada a partir de um arquivo de configuração no formato *JSON*, conforme ilustrado na figura 24. Essa documentação permanece disponível durante a execução do projeto.

Figura 23 – *Readme*

Fonte: elaborado pelo autor (2025).

[6] Disponível em https://swagger.io/
[7] Disponível em https://www.npmjs.com/package/swagger-ui-express

Figura 24 – Documentação *swagger*

Fonte: elaborado pelo autor (2025).

Como mostrado na figura 24, as rotas da API estão organizadas e documentadas por meio de *tags* definidas no código. Essas *tags* agrupam as rotas de acordo com suas funcionalidades específicas, facilitando a navegação e o entendimento da documentação. As rotas estão divididas nas seguintes categorias:

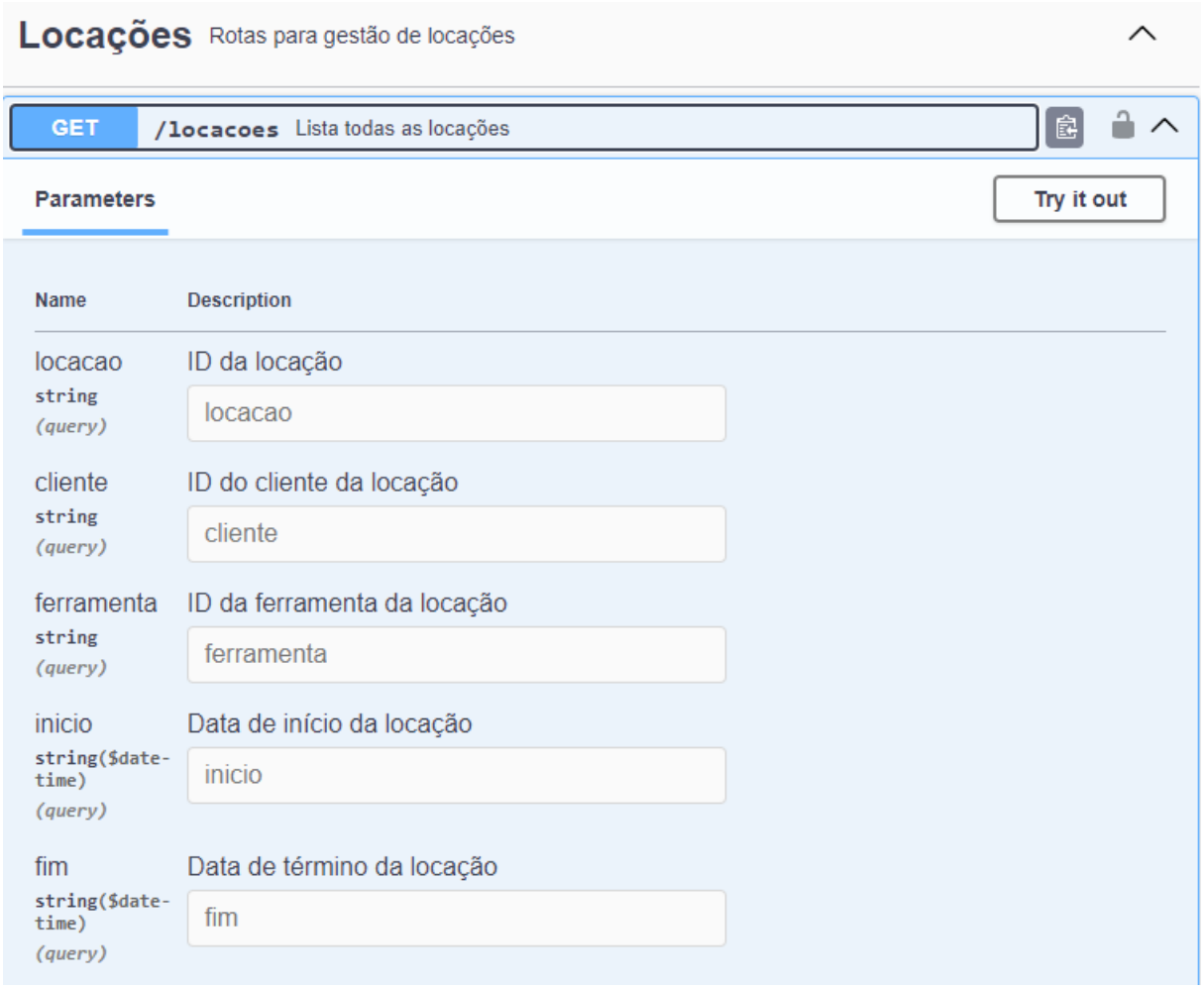
- Usuários: agrupa todas as rotas relacionadas à gestão de usuários.
- Grupos: contém as rotas responsáveis pela manipulação de grupos.
- Ferramentas: abrange as rotas que lidam com as operações sobre as ferramentas.
- Locações: reúne as rotas relacionadas ao gerenciamento das locações de ferramentas.
- Categorias: contém as rotas para gerenciamento das categorias de ferramentas.
- Clientes: agrupa as rotas referentes aos clientes da aplicação.
- Revisões: reúne as rotas que tratam das revisões de ferramentas.
- Rota: categoria que inclui as rotas de gestão de permissões.
- Autenticação: contém as rotas responsáveis pela autenticação de usuários.
- Recuperação de Senha: agrupa as rotas que permitem a recuperação de senha.

Embora a aplicação possua diversas rotas que desempenham funções essenciais, optou-se por mostrar, de forma objetiva, apenas a documentação da rota de locação, que é a principal responsável pelo gerenciamento das locações de ferramentas. As demais rotas, como usuário, grupos, ferramentas, categorias, clientes, revisões, autenticação e recuperação de senha, embora igualmente importantes, não serão detalhadas neste momento para manter o foco e a clareza na explicação da principal funcionalidade da aplicação.

A seguir, são apresentadas capturas de tela da documentação da rota de locação, incluindo exemplos das operações *GET, POST, PUT e DELETE*. As imagens demonstram os parâmetros de entrada, os métodos HTTP utilizados e exemplos de requisições e respostas.

- *GET*: rota que retorna todas as locações já existentes no banco de dados.

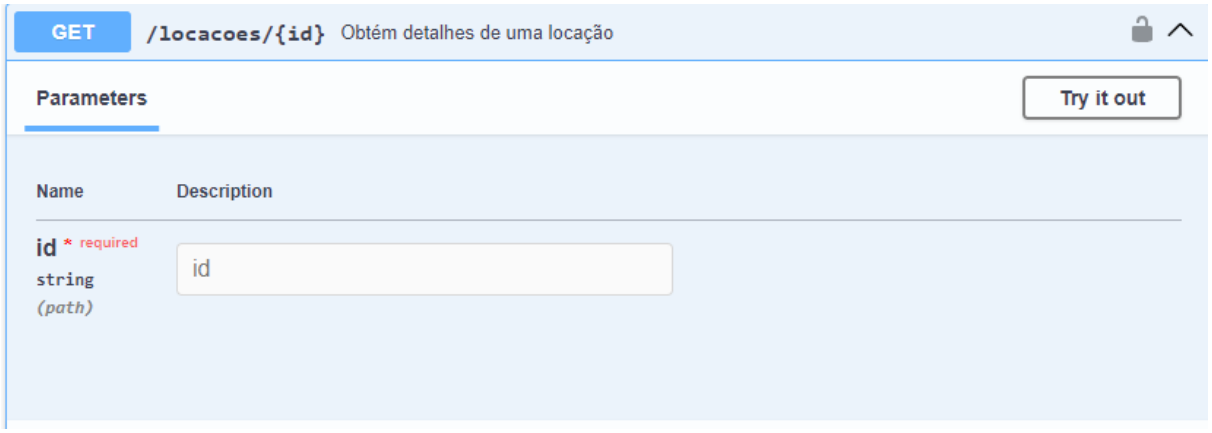
Figura 25 – Rota *GET*



Fonte: elaborado pelo autor (2025).

- *GET/id*: rota que retorna uma locação específica pelo *ID*.

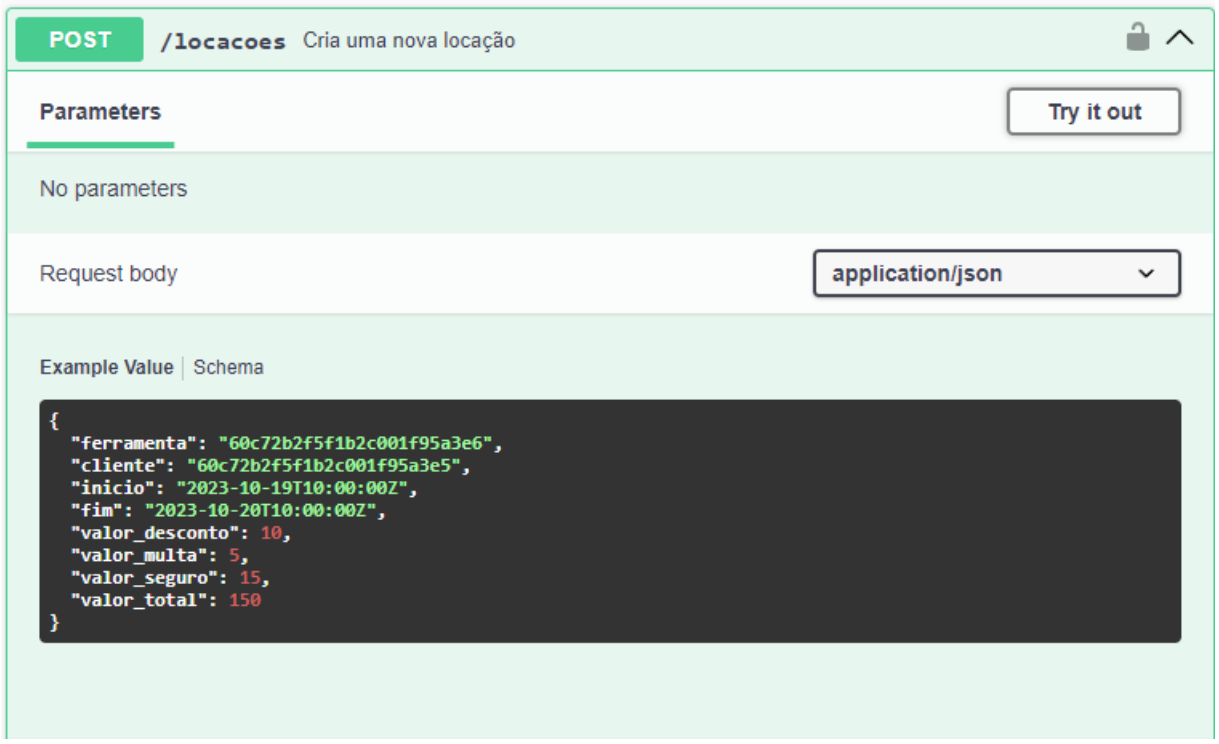
Figura 26 – Rota *GET/:id*



Fonte: elaborado pelo autor (2025).

- *POST*: rota que é responsável pelo cadastro de uma nova locação.

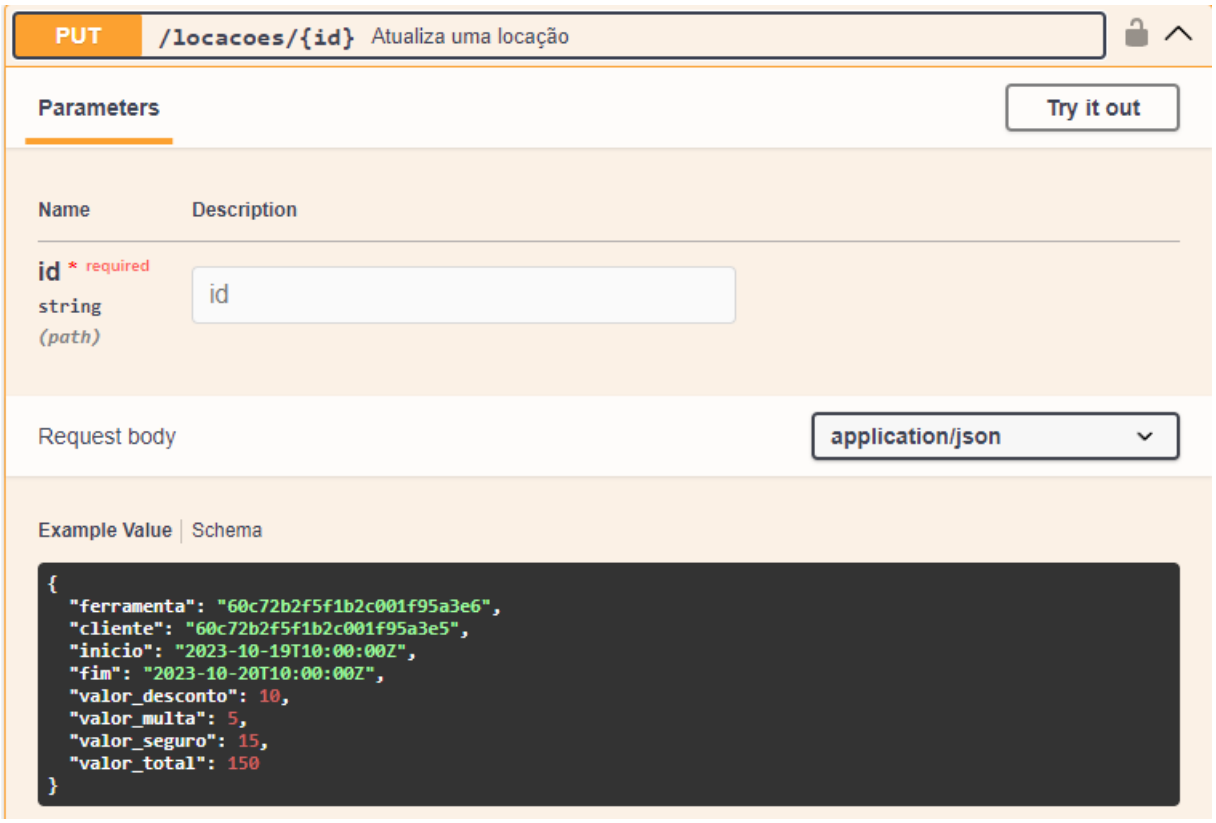
Figura 27 – Rota *POST*



Fonte: elaborado pelo autor (2025).

- *PUT*: rota que é responsável pela atualização de uma locação existente.

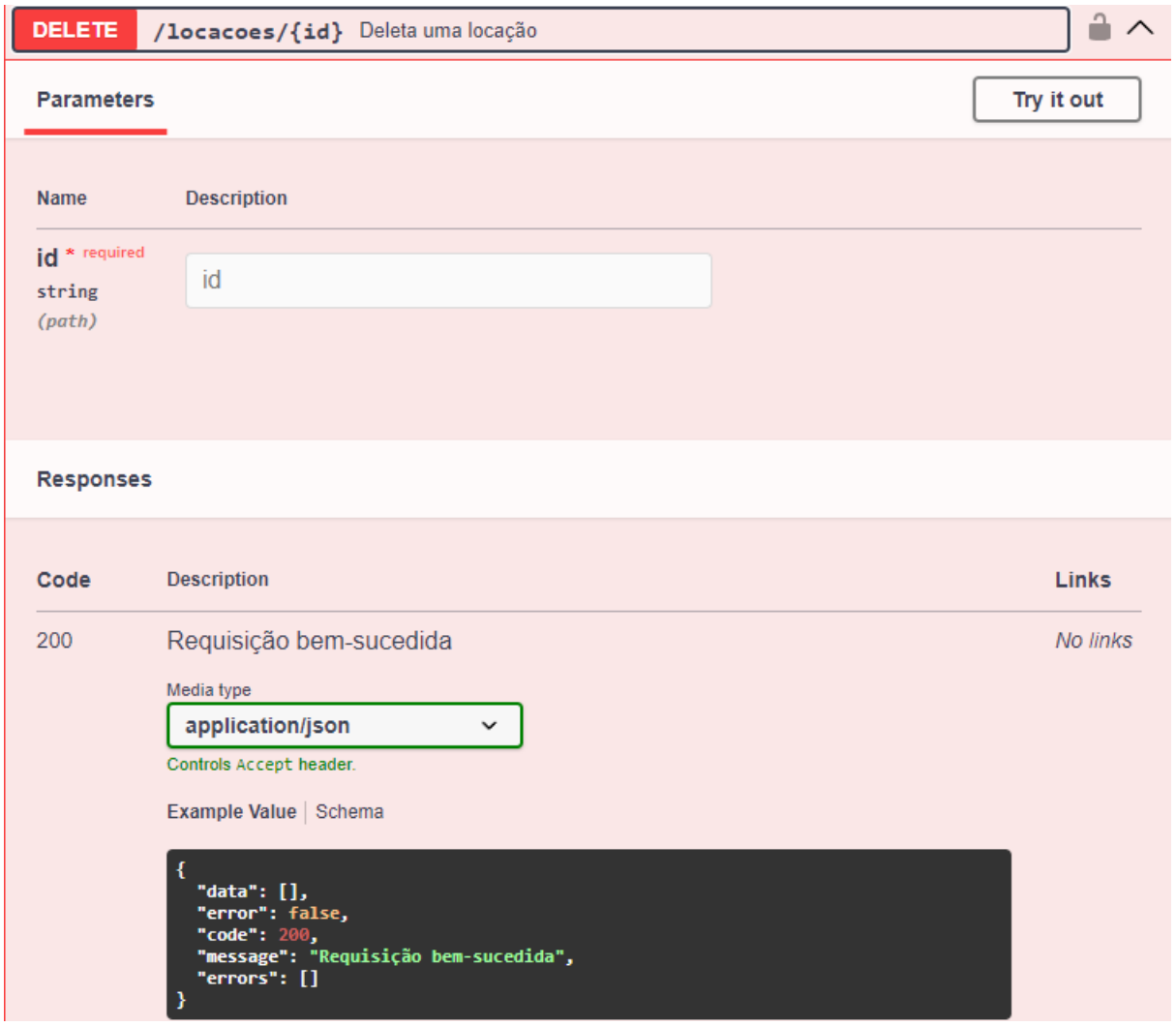
Figura 28 – Rota *PUT*



Fonte: elaborado pelo autor (2025).

- *DELETE*: rota que é responsável por excluir uma locação existente.

Figura 29 – Rota *DELETE*



Fonte: elaborado pelo autor (2025).

4.9 Implantação

A implantação da aplicação foi realizada por meio da técnica de conteinerização, com todos os serviços sendo executados em uma VM no servidor da FSLab [8].

4.10 Demonstração do software

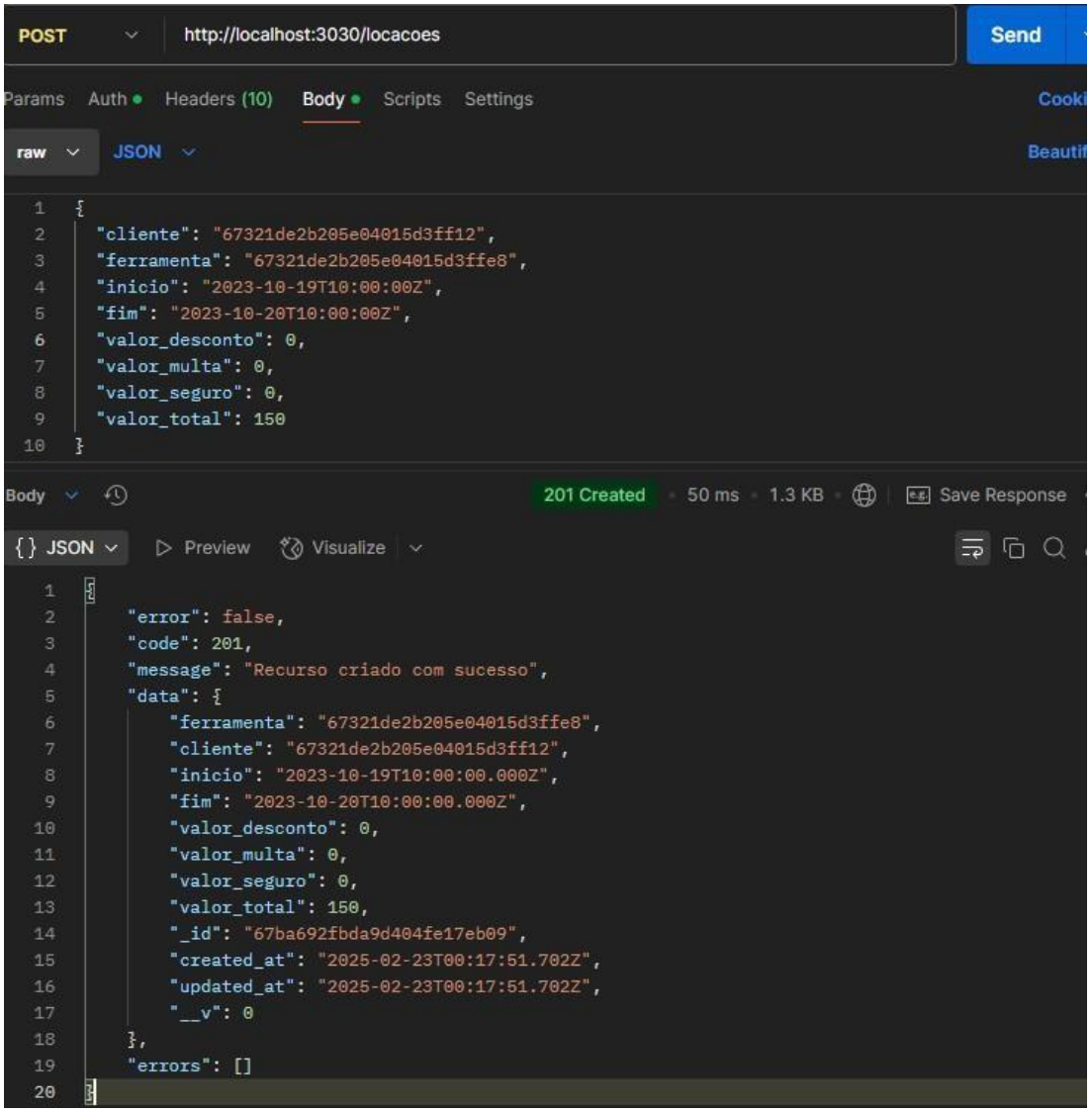
Para testar as funcionalidades da API, utilizou-se uma plataforma de requisições HTTP. Durante o desenvolvimento, foi empregado o *software Postman* [9], conforme mencionado anteriormente.

[8] Disponível em: https://api-gerenciamento.app.fslab.dev
[9] Disponível em https://www.postman.com/

Os testes das rotas podem ser realizados em https://api-gerenciamento.app.fslab.dev/. Para acessar, efetue o *login* com o e-mail *"admin@gmail.com"* e a senha *"Aa1@2024"* na rota *POST - /login*. A aplicação usa *hashing* de senha, garantindo maior segurança. Após o *login*, será retornado um *token* de acesso no cabeçalho das requisições para acessar outras funcionalidades da API.

O cadastro de locação é feito na rota *POST* - /locacoes. Para registrar uma locação, é necessário informar a identificação da ferramenta, a identificação do cliente, a data de início, a data de término, o desconto, o valor da multa, o valor do seguro e o valor total. Os *IDs* de ferramenta e cliente devem existir no banco de dados. Após o envio da requisição, se não houver inconsistências, a resposta será um status 201 *(Created)*, indicando sucesso no cadastro.

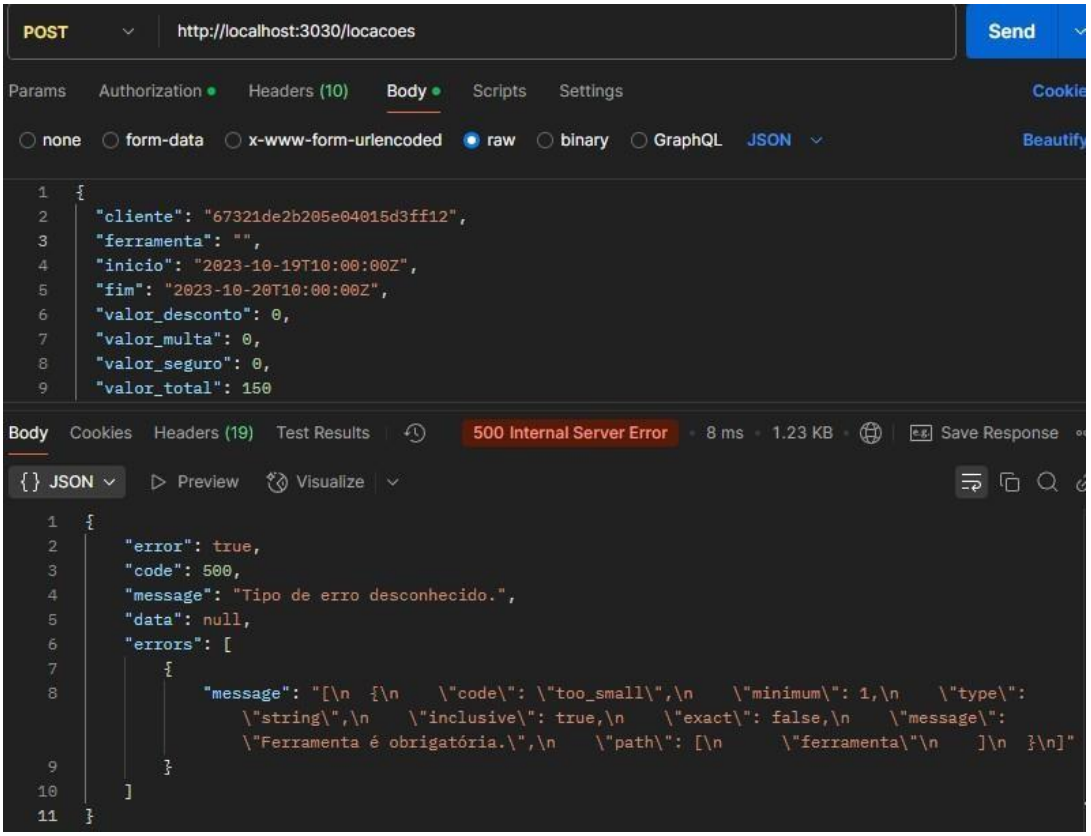
Figura 30 – *Created* Rota *POST*



Fonte: elaborado pelo autor (2025).

Em situações de erro interno, e de acordo com a natureza do problema no processamento da requisição, será retornado um erro exibindo os detalhes do problema. A figura 31 apresenta um exemplo de resposta de erro status 500 *(Internal Server Error)*.

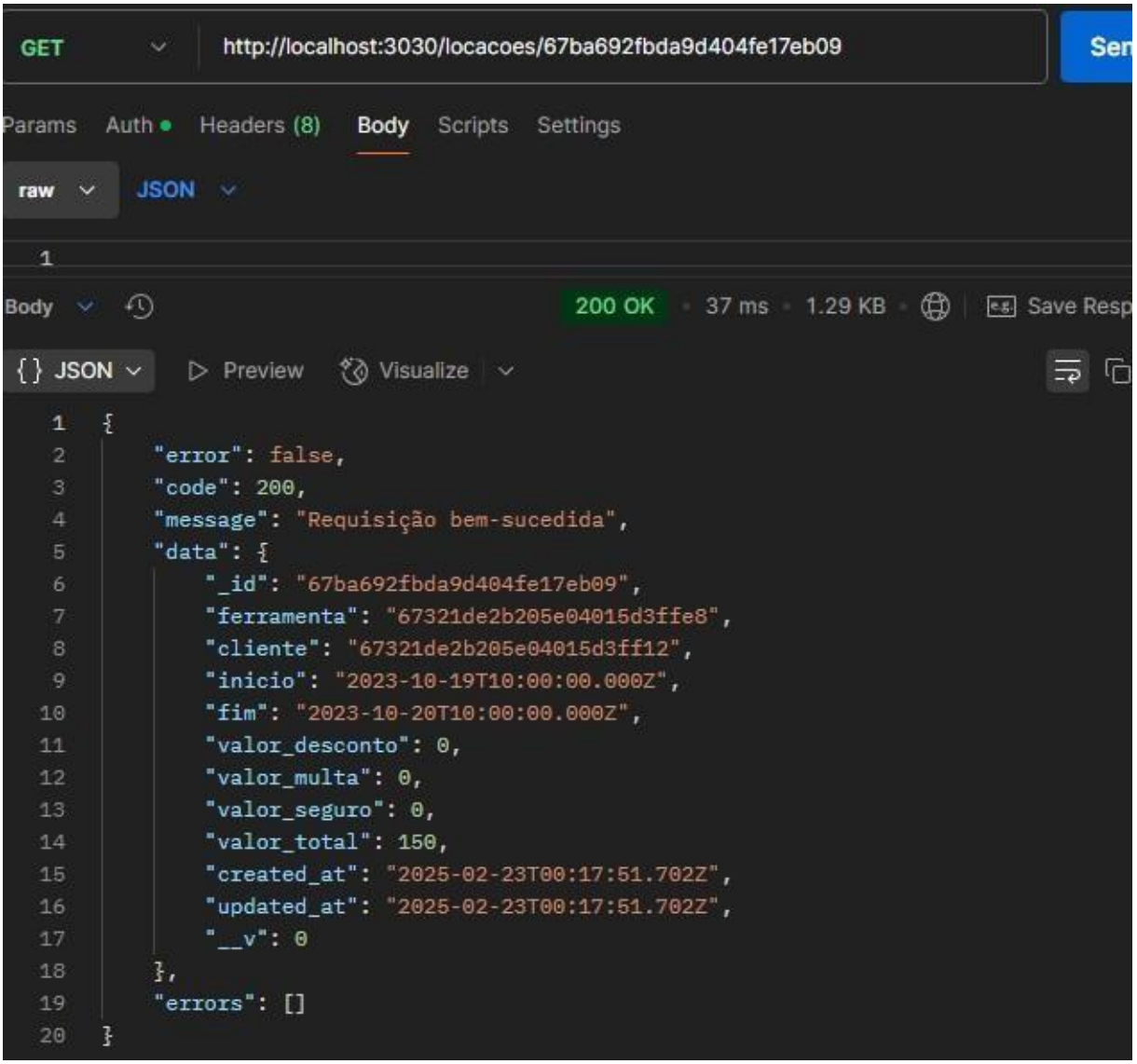
Figura 31 – Exemplo de erro 500 *(Internal Server Error)*



Fonte: elaborado pelo autor (2025).

Após a criação de uma locação, é possível visualizá-la na resposta de uma requisição na rota *GET* - "/locacoes/id", que retorna a locação cadastrada anteriormente no banco de dados, conforme demonstrado na figura 32 .

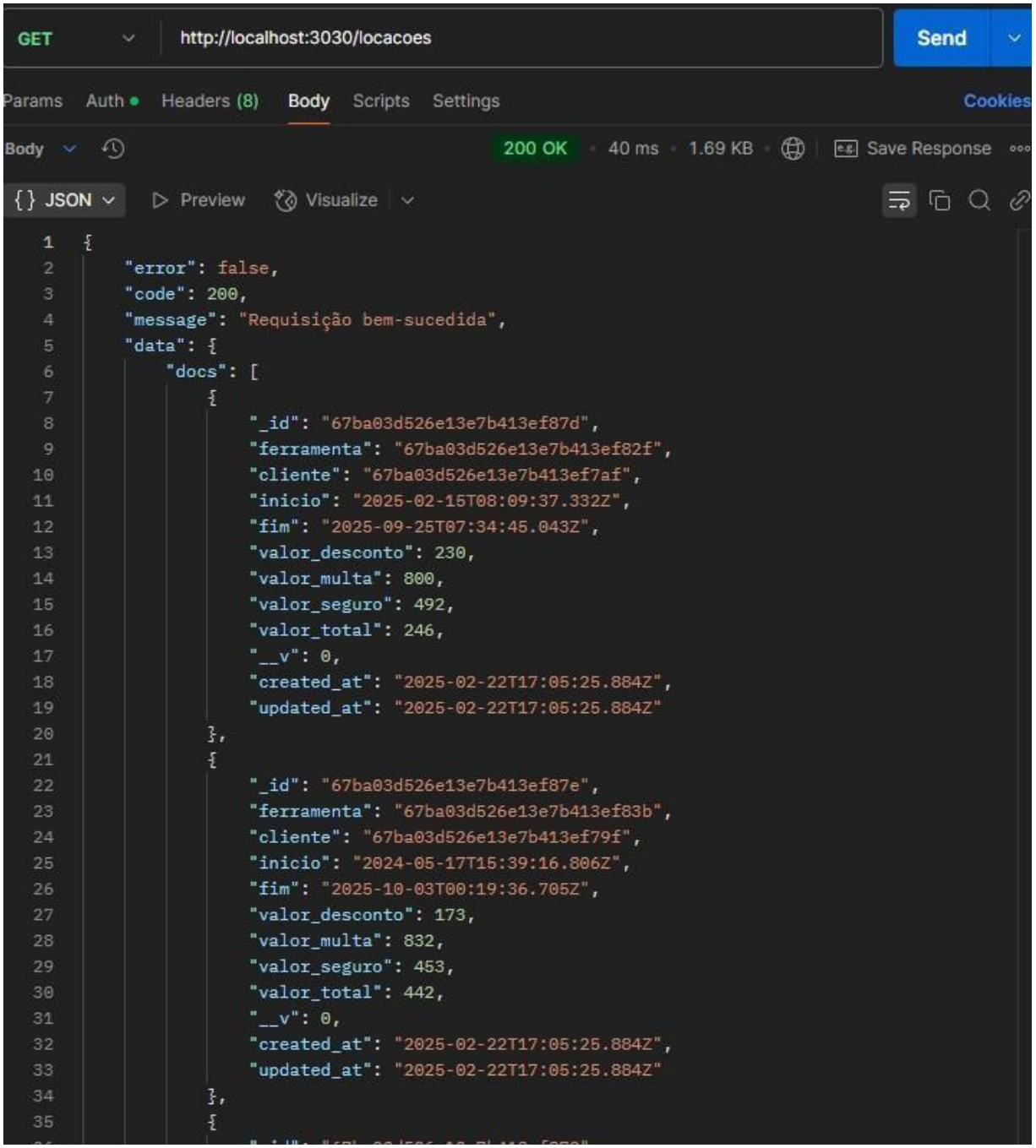
Figura 32 – Rota *GETid*



Fonte: elaborado pelo autor (2025).

Além disso, a rota *GET* - "/locacoes"retorna todas as locações cadastradas no banco de dados, com um sistema de paginação, permitindo o acesso às locações de forma organizada, conforme a figura 33.

Figura 33 – Rota *GET*

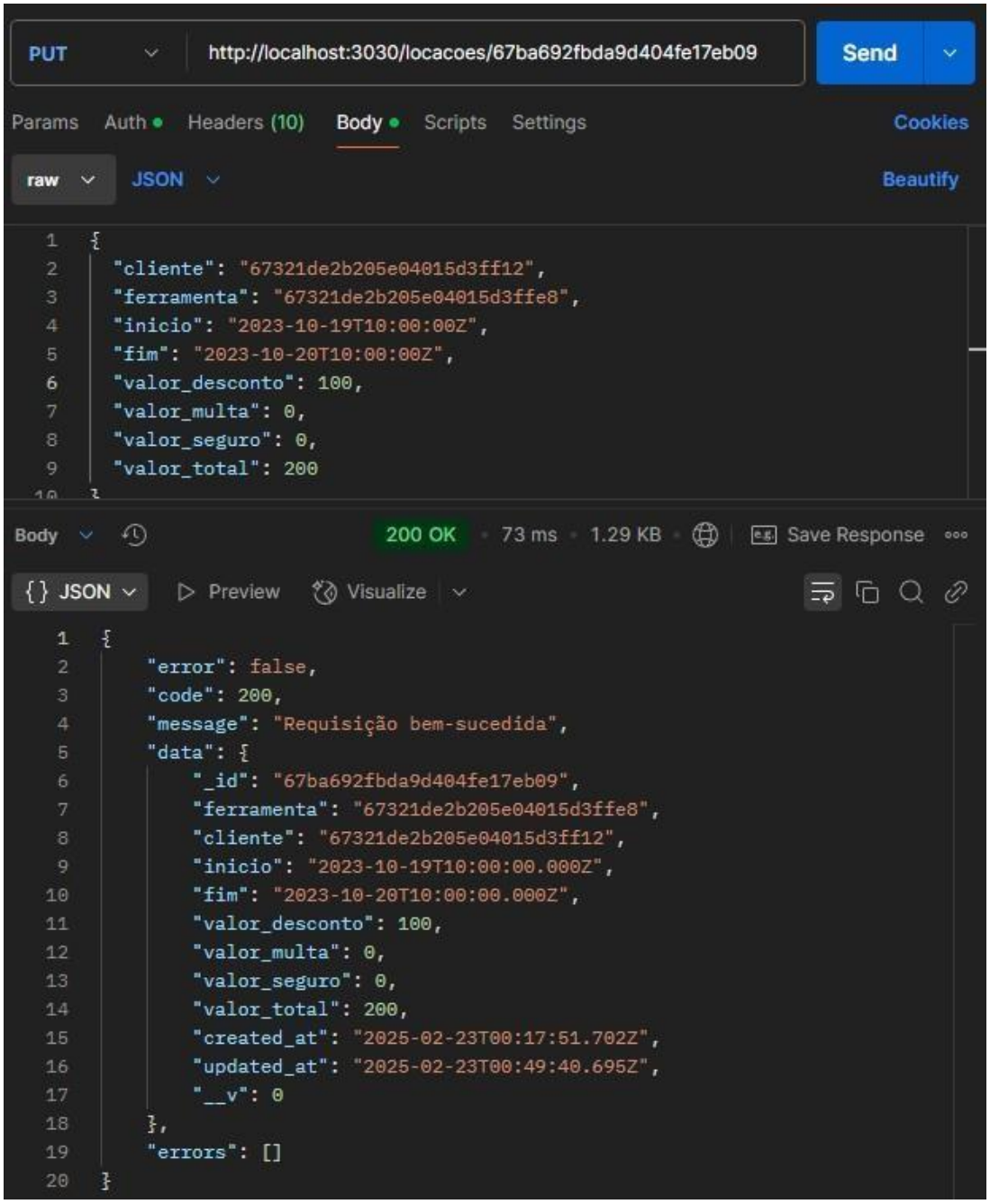


Fonte: elaborado pelo autor (2025).

A funcionalidade de atualização de locação está disponível na rota *PUT* - "/locacoes/*id*", permitindo modificar dados de locação previamente cadastrada. Para realizar a atualização, é necessário informar o *ID* da locação a ser alterada, juntamente com novos dados nos campos: identificação da ferramenta, identificação do cliente, data de início, data de término, desconto, valor da multa, valor do seguro e valor total. Vale ressaltar que todos os campos são opcionais, permitindo que o usuário altere apenas os dados desejados. Após

o envio da requisição, os dados são atualizados no banco de dados, resultando em uma resposta de sucesso com o *status* 200 (OK), caso não haja inconsistências. Caso contrário, a API retornará um erro detalhado, dependendo da natureza do problema, como mostrado nas mensagens de erro anteriores. A figura 34 apresenta a resposta de sucesso, indicando que a locação foi atualizada corretamente.

Figura 34 – Rota *PUTid*



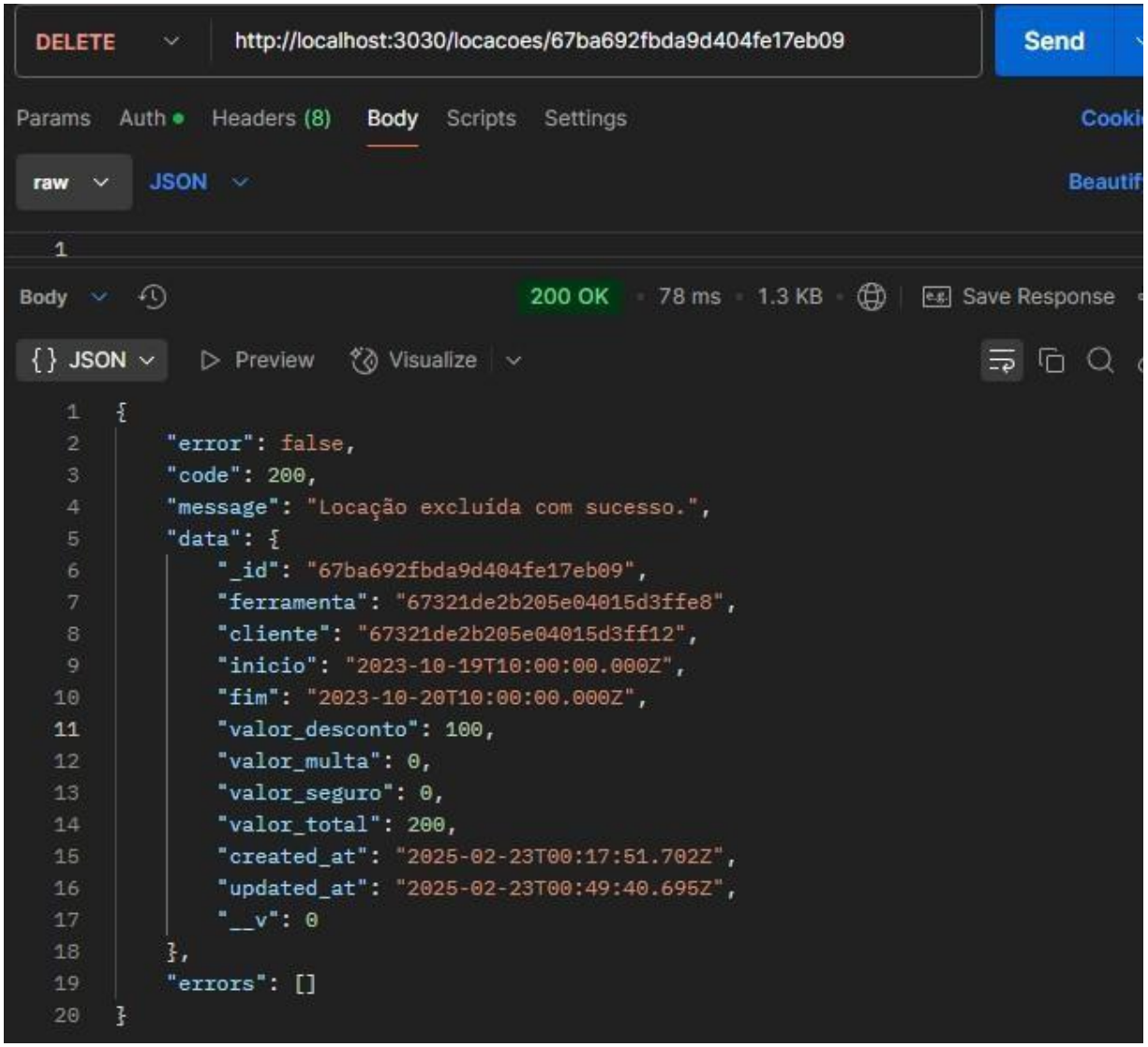
Fonte: elaborado pelo autor (2025).

Na funcionalidade de exclusão de locação disponível na rota *DELETE* - "/locacoes/*id*", permitindo remover a locação previamente cadastrada. Para realizar a exclusão,

é necessário informar o *ID* da locação que será removida.

Após envio da requisição, a locação será deletada do banco de dados, resultando em uma resposta de sucesso com *status* 200 (OK), caso a locação seja removida corretamente. A figura 35 apresenta a resposta de sucesso, indicando que a locação foi excluída corretamente.

Figura 35 – Rota *DELETEid*



Fonte: elaborado pelo autor (2025).

5 Considerações finais

Durante o desenvolvimento deste trabalho, passei por um grande aprendizado que transformou completamente minha forma de programar e de lidar com desafios tecnológicos. No início, eu tinha um conhecimento muito básico sobre as tecnologias utilizadas e não sabia usá-las de forma correta. Enfrentei algumas dificuldades, principalmente na construção da API, pois a estrutura inicial era simples e desorganizada.

Tudo começou a tomar outro rumo quando meu orientador sugeriu a adoção de uma arquitetura em camadas. Essa orientação foi essencial. A princípio, achei desafiador e complexo, mas, aos poucos, fui compreendendo os benefícios dessa abordagem. A mudança não só organizou melhor o código, como também aumentou a eficiência da aplicação. Essa transição foi um ponto de amadurecimento na minha jornada como desenvolvedora.

Outra etapa desafiadora foi a parte dos testes. Foram realizados mais de 800 testes para garantir o bom funcionamento da aplicação, e isso exigiu muito esforço da minha parte. Busquei, ao máximo, entregar uma solução funcional, com a maior cobertura possível, de forma que pelo menos 90% da aplicação estivesse funcionando corretamente. Esse processo foi muito desafiador para mim, pois eu não tinha muita experiência com testes automatizados. Passei bastante tempo tentando resolver as falhas, às vezes até mais tempo do que eu gostaria, mas, no final, aprendi muito com esses desafios.

Apesar das dificuldades enfrentadas ao longo do desenvolvimento deste trabalho de conclusão de curso, foi possível entregar uma solução funcional. Além de contribuir para minha evolução como desenvolvedora, essa experiência também fortaleceu minha capacidade de enfrentar desafios com resiliência, aprendendo com os erros e transformando obstáculos em oportunidades de crescimento.

Embora o desenvolvimento da API tenha sido uma grande conquista e tenha proporcionado um aprendizado valioso, é importante reconhecer as limitações do produto desenvolvido. Uma das principais limitações está na ausência de uma interface *front-end*, o que restringe a interação do usuário apenas ao nível de requisições da API. Isso pode tornar a utilização mais complexa para aqueles que não possuem familiaridade com ferramentas de teste de APIs, como o *Postman* [1] ou *Swagger* [2].

Outro ponto de limitação é a ausência de funcionalidades mais avançadas, como um sistema de notificação para alertar os administradores sobre locações pendentes ou em atraso, o que seria uma adição importante para o gerenciamento eficiente das locações. A integração com sistemas externos, como *gateways* de pagamento ou de emissão de nota fiscal eletrônica, ainda não foi implementada, mas são pontos que poderiam agregar valor

[1] Disponível em https://www.postman.com/
[2] Disponível em https://swagger.io/

à solução no futuro.

Apesar dessas limitações, a solução proposta atende ao escopo definido, proporcionando um sistema funcional para o gerenciamento de locações de ferramentas. Com isso, é possível concluir que o produto desenvolvido cumpre os objetivos iniciais, mas sempre há espaço para melhorias e evolução da aplicação.

5.1 Trabalhos futuros

Com o término do desenvolvimento da solução proposta, foram identificadas oportunidades de melhorias e expansões, incluindo:

- Desenvolvimento de um *Front-End*: Implementar uma interface gráfica para facilitar o uso da API por usuários não técnicos, aumentando a acessibilidade e usabilidade do sistema.
- Desenvolvimento de Aplicativo *Mobile*: Criar uma aplicação *mobile* para permitir o gerenciamento de locações diretamente de dispositivos móveis, proporcionando maior praticidade e acessibilidade para os administradores.
- Implementação de Rota de Estoque: Adicionar uma rota específica para o controle de estoque, permitindo um gerenciamento detalhado da quantidade disponível de cada ferramenta, além de registrar movimentações de entrada e saída.
- Sistema de Notificações: Integrar notificações para alertar sobre prazos de devolução, manutenção preventiva das ferramentas e mudanças no status das locações.
- Relatórios Gerenciais: Desenvolver relatórios personalizados para análise de métricas, como frequência de locações, faturamento, desempenho de ferramentas e histórico de revisões.
- Integração com *Gateways* de Pagamento: Facilitar o pagamento das locações diretamente pela aplicação, aceitando cartões de crédito, débito, Pix e boletos, de maneira segura e prática.
- Extração de Nota Fiscal: Automatizar a emissão de notas fiscais após a conclusão de uma locação, integrando com serviços de emissão fiscal.

Referências

ANDERSON, D.; REINERTSEN, D.; PINTO, A. *Kanban: Mudança Evolucionária de Sucesso Para Seu Negócio de Tecnologia*. [S.l.]: Roxby Media Limited, 2011. v. 6. ISBN 9780984521463. Citado na página 27.

CHODOROW, K. *MongoDB: The Definitive Guide*. 2. ed. Sebastopol, CA: O'Reilly Media, 2013. ISBN 978-1-449-34468-9. Citado na página 26.

EXPRESSJS. *Express: Web Application Framework*. 2025. https://expressjs.com/. Citado na página 32.

FLANAGAN, D. *JavaScript: o guia definitivo*. [S.l.]: Bookman Editora, 2012. Citado na página 26.

GUEDES, G. T. A. *UML 2: Uma abordagem prática*. [S.l.]: Novatec Editora, 2018. Citado 2 vezes nas páginas 34 e 40.

HOLMES, S. *Mongoose for Application Development*. [S.l.]: Packt Publishing Ltd, 2013. Citado na página 33.

JIN, B.; SAHNI, S.; SHEVAT, A. *Designing Web APIs: Building APIs That Developers Love*. [S.l.]: "O'Reilly Media, Inc.", 2018. Citado na página 25.

KANBANIZE. *O que é Kanban?* 2024. Disponível em: https://kanbanize.com/pt/recursos-kanban/primeiros-passos/o-que-e-kanban. Citado na página 27.

LAUDON, K. C.; LAUDON, J. P. *Sistemas de Informação Gerenciais*. 14. ed. São Paulo: Pearson, 2019. ISBN 978-85-221-3616-2. Citado na página 23 e 24.

LOELIGER, J.; MCCULLOUGH, M. *Version Control with Git: Powerful tools and techniques for collaborative software development*. [S.l.]: "O'Reilly Media, Inc.", 2012. Citado na página 47.

Mozilla Developer Network. *JavaScript - MDN Web Docs*. 2023. Acesso em: 23 fev. 2025. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Citado na página 26.

NGUYEN, D. *Jump Start Node. js: Get Up to Speed With Node. js in a Weekend*. [S.l.]: SitePoint Pty Ltd, 2012. Citado na página 32.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: uma abordagem profissional*. 8. ed. Porto Alegre: AMGH Editora, 2016. Citado na página 29.

SCHROEDER, F. d. S. R.; SANTOS, F. dos. *Arquitetura e testes de serviços web de alto desempenho com Node.js e MongoDB*. Citado na página 26.

TURBAN, E.; VOLONINO, L.; WOOD, G. *Tecnologia da Informação para Gestão: Transformando os Negócios na Era Digital*. 10. ed. São Paulo: Cengage Learning, 2018. ISBN 978-85-221-3463-2. Citado na página 23.

Anexos

ANEXO A – Licença MIT