INSTITUTO FEDERAL GOIANO
CAMPUS URUTAÍ
NÚCLEO DE INFORMÁTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

DIEGO RIBEIRO ARAÚJO
FELIPE RIBEIRO ARAÚJO

DESENVOLVIMENTO DE UMA API PARA GERENCIAMENTO DE *TICKETS* DE SERVIÇOS

Urutaí, 18 de julho de 2025

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

Araújo, Diego Ribeiro
A663d Desenvolvimento de uma API para gerenciamento de tickets de serviços / Diego Ribeiro Araújo. Urutaí 2025.

60f. il.

Orientadora: Profª. Dra. Cristiane de Fatima dos Santos Cardoso.
Tcc (Bacharel) - Instituto Federal Goiano, curso de 0120201 - Bacharelado em Sistemas de Informação - Urutaí (Campus Urutaí).
1. Spring Boot, API REST, Gerenciamento de Tickets, JSON Web Token (JWT). 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 9.610/98, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia Goiano, a disponibilizar gratuitamente o documento no Repositório Institucional do IF Goiano (RIIF Goiano), sem ressarcimento de direitos autorais, conforme permissão assinada abaixo, em formato digital para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IF Goiano.

Identificação da Produção Técnico-Científica

[ ] Tese	[ ] Artigo Científico
[ ] Dissertação	[ ] Capítulo de Livro
[ ] Monografia – Especialização	[ ] Livro
[X] TCC - Graduação	[ ] Trabalho Apresentado em Evento

[ ] Produto Técnico e Educacional - Tipo: _____

Nome Completo do Autor: Diego Ribeiro Araújo, Felipe Ribeiro Araújo
Matrícula: 2022101202010058, 2020101202010073
Título do Trabalho:Desenvolvimento de uma API para gerenciamento de tickets de serviços

Restrições de Acesso ao Documento

Documento confidencial: [X] Não [ ] Sim, justifique: _____

_____

Informe a data que poderá ser disponibilizado no RIIF Goiano: 17/07/2025
O documento está sujeito a registro de patente? [ ] Sim [X] Não
O documento pode vir a ser publicado como livro? [ ] Sim [X] Não

DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O/A referido/a autor/a declara que:

1. 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;
2. obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autor/a, 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;
3. 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.

gov.br Documento assinado digitalmente
DIEGO RIBEIRO ARAUJO
Data: 17/07/2025 20:23:25-0300
Verifique em https://validar.iti.gov.br

gov.br Documento assinado digitalmente
FELIPE RIBEIRO ARAUJO
Data: 17/07/2025 21:17:21-0300
Verifique em https://validar.iti.gov.br

Urutaí , 17/ 07/2025.
Local Data

_____
Assinatura do Autor e/ou Detentor dos Direitos Autorais

Ciente e de acordo:

Documento assinado digitalmente
CRISTIANE DE FATIMA DOS SANTOS CARDOSO
Data: 17/07/2025 23:12:10-0300
Verifique em https://validar.iti.gov.br

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

Formulário 14/2025 - CCBSI-URT/GE-UR/DE-UR/CMPURT/IFGOIANO

INSTITUTO FEDERAL GOIANO – CAMPUS URUTAÍ

DIRETORIA / GERÊNCIA DE GRADUAÇÃO E PÓS-GRADUAÇÃO

COORDENAÇÃO DOS CURSOS DA ÁREA DE INFORMÁTICA

CURSO SUPERIOR DE SISTEMAS DE INFORMAÇÃO

ATA DE APRESENTAÇÃO DE TRABALHO DE CURSO

Aos 26 dias do mês de junho de dois mil e vinte e cinco, reuniram-se os professores: Cristiane de Fátima dos Santos Cardoso, Jean Tomaz da Silva e Nattane Luiza da Costa no núcleo de Informática do Instituto Federal Goiano - Campus Urutaí, para avaliar o Trabalho de Curso do(s) acadêmico(s): **DIEGO RIBEIRO ARAÚJO e FELIPE RIBEIRO ARAÚJO**, como requisito necessário para a conclusão do Curso Superior de Sistemas de Informação desta Instituição. O presente TC tem como título: **DESENVOLVIMENTO DE UMA API PARA GERENCIAMENTO DE TICKETS DE SERVIÇOS,** foi orientado por Cristiane de Fátima dos Santos Cardoso.

Após análise, foram dadas as seguintes notas:

Professores	**Aluno / Notas**	
	DIEGO RIBEIRO ARAÚJO	FELIPE RIBEIRO ARAÚJO
1. Cristiane de Fátima dos Santos Cardoso	6,5	6,5
2. Jean Tomaz da Silva	9,5	9,5
3. Nattane Luiza da Costa	9	9

MÉDIA FINAL:	8,33	8,33

OBSERVAÇÕES: Realizar correções sugeridas pela banca e submeter o trabalho a nova avaliação por parte do orientador até 06/08/2025. A versão final deve ser depositada no RIIF GOIANO (Repositório Institucional do IF Goiano) até o dia 23/08/2025, conforme tutorial https://www.youtube.com/watch?v=l8h2mNilaMA

Por ser verdade firmamos a presente.

Documento assinado eletronicamente por:

- **Cristiane de Fatima dos Santos Cardoso** , **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 26/06/2025 12:04:35.
- **Nattane Luiza da Costa**, **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 27/06/2025 09:17:55.
- **Jean Tomaz da Silva**, **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 30/06/2025 10:35:01.

Este documento foi emitido pelo SUAP em 26/06/2025. 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: 720375
Código de Autenticação: 822ccdbbdf

INSTITUTO FEDERAL GOIANO
Campus Urutaí
Rodovia Geraldo Silva Nascimento, Km 2.5, SN, Zona Rural, URUTAÍ / GO, CEP 75790-000
(64) 3465-1900

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

Formulário 13/2025 - CCBSI-URT/GE-UR/DE-UR/CMPURT/IFGOIANO

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA GOIANO

CURSO DE SISTEMAS DE INFORMAÇÃO

DESENVOLVIMENTO DE UMA API PARA GERENCIAMENTO DE TICKETS DE SERVIÇOS

Autores: DIEGO RIBEIRO ARAÚJO e

FELIPE RIBEIRO ARAÚJO

Orientador(a): Cristiane de Fátima dos Santos Cardoso

Monografia defendida por DIEGO RIBEIRO ARAÚJO e FELIPE RIBEIRO ARAÚJO, e aprovada em 26 de junho de 2025, pela banca examinadora constituída pelos membros:

APROVADA em 26 de junho de 2025

Profª. Drª. Cristiane de Fátima dos Santos Cardoso

Presidente da Banca

Profª. Me. Jean Tomaz da Silva

Avaliador

Profª. Drª. Nattane Luiza da Costa

Avaliadora

Documento assinado eletronicamente por:

- **Cristiane de Fatima dos Santos Cardoso** , **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 26/06/2025 11:10:16.
- **Nattane Luiza da Costa**, **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 26/06/2025 11:59:59.
- **Jean Tomaz da Silva**, **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 30/06/2025 10:37:15.

Este documento foi emitido pelo SUAP em 26/06/2025. 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: 720378
Código de Autenticação: deb67182bc

INSTITUTO FEDERAL GOIANO
Campus Urutaí
Rodovia Geraldo Silva Nascimento, Km 2.5, SN, Zona Rural, URUTAÍ / GO, CEP 75790-000
(64) 3465-1900

AGRADECIMENTOS

Agradeço primeiramente a Deus, pela força, saúde e sabedoria concedidas durante toda minha caminhada acadêmica. Aos meus pais e familiares, pelo amor, apoio incondicional e incentivo contínuo em todas as etapas da minha vida.Aos colegas de turma e amigos que estiveram presentes durante essa jornada, pelas trocas de experiências, pelas conversas e pelas colaborações que tornaram essa caminhada mais leve e significativa. E, por fim, agradeço à instituição pela estrutura e oportunidade de aprendizado que me proporcionou ao longo da graduação.

RESUMO

O presente Trabalho de Conclusão de Curso tem como objetivo o desenvolvimento de uma API para gerenciamento de *tickets* de atendimento, oferecendo uma solução eficiente, flexível e customizável para empresas que necessitam organizar e otimizar seus fluxos de suporte ao cliente. O trabalho justifica-se pela crescente demanda por ferramentas que melhorem a comunicação entre cliente e suporte, reduzindo falhas operacionais e promovendo maior agilidade e controle nos atendimentos. A metodologia adotada para o desenvolvimento com foco em entregas funcionais, utilizando a linguagem Java com o framework Spring Boot, banco de dados MySQL, autenticação via JWT, cache e documentação da API por meio do Swagger. A API implementa funcionalidades como autenticação segura via JWT, notificações automáticas, fluxos inteligentes de triagem e acompanhamento de chamados, além de ser construída com uma arquitetura modular e escalável, favorecendo a manutenção e a integração com outros sistemas corporativos. A solução proposta visa contribuir diretamente para a satisfação do cliente e para a eficiência dos processos internos das empresas.

Palavras-chave:

Spring Boot, API REST, Gerenciamento de *Tickets*, JSON Web Token (JWT)

LISTA DE FIGURAS

3.1 Diagrama de casos de uso 27
3.2 Diagrama de sequência do usuário 30
3.3 Diagrama de sequência da Regra de Prioridade 32
3.4 Diagrama de classe 33

4.1 Diagrama de classes do banco de dados 38
4.2 Estrutura de pacotes do Back-end 39
4.3 Visualização geral da interface do Swagger 57
4.4 Endpoints disponíveis na Api 57
4.5 Requisição de login 58
4.6 Resposta da requisição de login 58

5.1 Criar Usuário 60
5.2 Erros na criação de usuários 61
5.3 Realizar Login 61
5.4 Criar *Ticket* 62
5.5 Registrar atendimento de algum *ticket* 63
5.6 Criar Comentário de um *ticket* 64

LISTA DE TABELAS

2.1 Comparativo de Características em Sistemas de Gerenciamento de *Tickets* . . . 21

3.1 Tabela de requisitos funcionais . . . 25
3.2 Descrição do Caso de Uso UC01 - Fazer Cadastro . . . 28
3.3 Descrição do Caso de Uso UC02 - Gerenciar *Ticket* . . . 29
3.4 Descrição do Caso de Uso UC03 - Fazer Comentário . . . 29
3.5 Tabela consolidada de tecnologias, aspectos e descrições . . . 34

4.1 Tabela de dependências utilizadas . . . 36
4.2 Endpoints da API de Autenticação . . . 40
4.3 Endpoints da API de Usuários . . . 40
4.4 Endpoints da API de Categorias . . . 41
4.5 Endpoints da API de Departamentos . . . 41
4.6 Endpoints da API de Regras de Prioridade . . . 42
4.7 Endpoints da API de *Tickets* . . . 42
4.8 Endpoints da API de Registros de Trabalho . . . 43
4.9 Endpoints da API de Comentários . . . 43

5.1 Comparativo de Características em Sistemas de Gerenciamento de *Tickets* para o sistema criado . . . 66

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface* ou Interface de Programação de Aplicações. Pag.17

JWT *JSON Web Token*. Pag.16

REST *Representational State Transfer* ou Transferência de Estado Representacional. Pag.20

SLA *Service Level Agreement*, ou Acordo de Nível de Serviço. Pag.15

SUMÁRIO

1 INTRODUÇÃO **15**

2 Fundamentos Teóricos **17**
2.1 Serviços disponibilizados por meio de *Tickets* 17
2.1.1 Exemplo de Serviço: Suporte Técnico 17
2.1.2 Descrição do Processo de Gerenciamento de *Tickets* 18
2.2 Atendimento ao Cliente e Sistemas de Suporte 19
2.3 APIs e Integração de Sistemas 19
2.4 Segurança e Autenticação 20
2.5 Comparação com Soluções Existentes 21
2.6 Desenvolvimento de APIs RESTful 22

3 Levantamento e Análise de requisitos **23**
3.1 Levantamento e Análise de Requisitos 23
3.1.1 Níveis de acesso do Usuário 24
3.1.2 Requisitos Funcionais 24
3.1.3 Regras de Negócio 25
3.1.4 Envio de E-mails 26
3.1.5 Diagrama de Casos de Uso 27
3.1.6 Diagrama de Sequência 29
3.1.7 Diagrama de Classes de domínio 32
3.2 Definição de Tecnologias e Ferramentas 34

4 Implementação do Back-end (API REST) **35**
4.1 Configuração Inicial 35
4.2 Projeto do Banco de Dados 37
4.3 Estrutura de Pacotes do Back-end 38
4.4 Endpoints da api 39
4.4.1 Endpoints de Autenticação 39
4.4.2 Endpoints de Usuários 40
4.4.3 Endpoints de Categorias 40
4.4.4 Endpoints de Departamentos 41
4.4.5 Endpoints de Regras de Prioridade 41
4.4.6 Endpoints de *Tickets* 42

4.4.7 Endpoints de Registros de Trabalho 42
4.4.8 Endpoints de Comentários 43
4.5 Trechos de Código 43
4.5.1 Usuário 44
4.5.2 Regra Prioridade 45
4.5.3 Departamento 47
4.5.4 *Ticket* 48
4.5.5 Registro de Trabalho 55
4.6 Documentação Swagger do sistema 56

5 A aplicação **59**
5.1 Criar Usuário 59
5.2 Realizar Login 61
5.3 Criar *Ticket* 62
5.4 Registrar Atendimento 62
5.5 Criar Comentário 63

CONCLUSÃO **65**

CAPÍTULO 1

INTRODUÇÃO

Nos dias atuais, a eficiência no atendimento ao cliente é um fator determinante para o sucesso das empresas, independentemente do seu porte ou segmento. Problemas técnicos, dúvidas e solicitações de melhorias são recorrentes no cotidiano organizacional, exigindo um gerenciamento estruturado para garantir que essas demandas sejam resolvidas de maneira ágil e eficiente.

Nesse contexto, sistemas de gerenciamento de *tickets* se tornam ferramentas essenciais para a organização e otimização do suporte ao cliente. Estudos recentes demonstram que a centralização e a otimização dos processos de atendimento colaboram significativamente para a satisfação e fidelização dos clientes (FITZSIMMONS; FITZSIMMONS, 2023). Segundo Axelos (2019), a implementação de boas práticas de gerenciamento de serviços pode reduzir falhas operacionais e aumentar a produtividade das equipes de suporte.

Alguns sistemas comerciais para gerenciamento de *tickets* amplamente utilizados no mercado incluem o Zendesk, Freshdesk e Zoho Desk. O Zendesk (2007) é uma das soluções mais conhecidas e utilizadas mundialmente, oferecendo uma plataforma robusta com funcionalidades como automações avançadas, base de conhecimento integrada, relatórios personalizáveis e suporte omnichannel (e-mail, chat, redes sociais). Já o Freshdesk (2010) se destaca por sua interface intuitiva e pela oferta de planos acessíveis, sendo ideal para pequenas e médias empresas que desejam funcionalidades como SLA (*Service Level Agreement*, ou Acordo de Nível de Serviço) é um compromisso formal que define os prazos e condições para a resolução de tickets ou a prestação de um serviço. Em sistemas de atendimento, o SLA determina, por exemplo, o tempo máximo para a resposta ou solução de um chamado com base em sua prioridade,

garantindo maior previsibilidade e controle de qualidade no suporte ao cliente. O (Zoho Desk, 2010), por sua vez, é uma ferramenta que se integra de forma nativa com o ecossistema da Zoho, proporcionando gestão de *tickets* com foco em produtividade, acompanhamento por canais diversos e automação inteligente de processos.

O sistema proposto neste trabalho apresenta um diferencial estratégico em relação a essas ferramentas genéricas, ao oferecer uma solução customizável, adaptável às particularidades de cada empresa. Destacam-se recursos como a definição flexível de fluxos de atendimento, categorização personalizada de *tickets*, configuração de níveis de prioridade com base em regras específicas, além da possibilidade de integração com sistemas já existentes. Com isso, a aplicação torna-se uma alternativa eficiente para organizações que desejam um sistema sob medida, sem depender exclusivamente de ferramentas comerciais com escopo fixo.

É importante destacar que a solução prioriza segurança e controle, implementando autenticação baseada em JWT (*JSON Web Token*) para garantir acessos seguros e evitar manipulação indevida dos *tickets*, alinhando-se às práticas modernas de segurança recomendadas por estudos recentes (STALLINGS, 2023). A arquitetura do sistema foi projetada para ser modular e altamente eficiente, permitindo fácil manutenção e escalabilidade vertical, assegurando um alto desempenho mesmo com um grande volume de chamados.

O desenvolvimento dessa API se justifica pela necessidade crescente das empresas em aprimorar seus processos internos de atendimento, tornando-os mais ágeis e eficientes. A ausência de um sistema adequado pode resultar em atendimentos desorganizados, perda de informações e demora na resolução dos chamados, impactando negativamente a satisfação dos clientes (TALEBI; BARDSIRI, 2023). Com a implementação de um sistema estruturado e customizável, busca-se minimizar esses problemas, proporcionando uma gestão eficaz dos *tickets* e contribuindo para a melhoria contínua dos serviços prestados.

Diante desse cenário, o objetivo deste trabalho é desenvolver uma API eficiente e flexível para o gerenciamento de *tickets* de serviço, permitindo que empresas centralizem e otimizem o registro, acompanhamento e resolução de chamados. Com isso, busca-se melhorar a experiência dos usuários, reduzir gargalos no atendimento e garantir um fluxo de suporte mais ágil e organizado.

Nos capítulos seguintes, serão apresentados os fundamentos teóricos relacionados ao gerenciamento de *tickets*, as tecnologias utilizadas no desenvolvimento da API, a modelagem da solução e os resultados obtidos.

CAPÍTULO 2

FUNDAMENTOS TEÓRICOS

Esta seção apresenta os conceitos e bases teóricas que fundamentam o desenvolvimento da API (*Application Programming Interface* ou Interface de Programação de Aplicações) para gerenciamento de *tickets* de serviço. São abordados os temas de atendimento ao cliente, integração de sistemas por meio de APIs, práticas de segurança e as limitações das soluções tradicionais, evidenciando como esses conhecimentos orientam as escolhas do projeto.

2.1 Serviços disponibilizados por meio de *Tickets*

Nesta seção, será ilustrado um serviço de suporte técnico, onde os usuários podem abrir *tickets* para solicitar assistência em relação a problemas técnicos em um produto ou serviço oferecido por uma empresa. O sistema de gerenciamento de *tickets* permite que os usuários classifiquem os chamados com base em categorias e prioridades, facilitando o direcionamento adequado e a resolução eficiente dos problemas.

2.1.1 Exemplo de Serviço: Suporte Técnico

Suponha que um cliente enfrenta dificuldades técnicas com um software oferecido por uma empresa. O cliente cria um *ticket* por meio do sistema, e o serviço de suporte técnico começa a agir com base em categorias e prioridades predefinidas.

Categorização de *Tickets*

Os tickets podem ser classificados em diferentes categorias para garantir que a equipe de suporte atenda à solicitação corretamente e de forma rápida. Algumas categorias possíveis

incluem:

- **Problema Técnico:** Relacionado a falhas de sistema, bugs ou erros no software.
- **Pedido de Melhoria:** Solicitações para novas funcionalidades ou melhorias no produto.
- **Dúvida de Uso:** Questões relacionadas ao uso do sistema ou dúvidas de configuração.
- **Solicitação de Suporte:** Pedido de assistência em tarefas específicas, como instalação ou configuração.

Prioridades dos *tickets*

A prioridade de cada *ticket* pode ser definida com base na urgência e impacto do problema. Exemplos de níveis de prioridade podem incluir:

- **Alta:** O *ticket* refere-se a um problema crítico, que impacta diretamente o funcionamento do sistema ou da operação do cliente.
- **Média:** O *ticket* se refere a um problema significativo, mas não crítico, que pode ser resolvido com um pouco mais de tempo.
- **Baixa:** O *ticket* se refere a um problema menor ou a uma dúvida que não impacta significativamente a operação ou o uso do sistema.

2.1.2 Descrição do Processo de Gerenciamento de *Tickets*

O processo de gerenciamento de *tickets* inicia com a criação de um *ticket* pelo cliente, onde ele especifica a categoria (ex: "Problema Técnico") e a prioridade (ex: "Alta"). Após o envio, o sistema encaminha o *ticket* para a equipe de suporte apropriada, de acordo com a categoria selecionada. O *ticket* será então classificado por um atendente que, dependendo da prioridade, tomará ações imediatas, como entrar em contato com o cliente para obter mais informações ou iniciar uma solução.

Os *tickets* de alta prioridade são atendidos primeiro, enquanto os de baixa prioridade são programados para resolução em prazos mais extensos. O agendamento das resoluções é feito com base na prioridade definida no momento da criação do ticket. Cada nível de prioridade está associado a um tempo máximo de resolução: por exemplo, tickets de alta prioridade devem ser solucionados em até 4 horas após a sua criação, enquanto tickets de baixa prioridade possuem

um prazo de até 3 dias corridos. Esses prazos servem como base para a organização da fila de atendimento e para o monitoramento dos acordos de nível de serviço (SLA). O processo de resolução, por sua vez, envolve a análise do problema, a execução das correções necessárias e a comunicação com o cliente sobre o andamento ou a solução final.

2.2 Atendimento ao Cliente e Sistemas de Suporte

A excelência no atendimento ao cliente é considerada um dos principais diferenciais competitivos no mercado atual. Empresas de diversos segmentos têm percebido que a satisfação do cliente está diretamente ligada à eficiência dos processos de suporte e à rapidez na resolução de problemas (FITZSIMMONS; FITZSIMMONS, 2023). Sistemas de gerenciamento de *tickets* são fundamentais para estruturar esse atendimento, pois permitem:

- **Centralização das demandas:** Reunindo todas as solicitações em um único ambiente, o que facilita o monitoramento e o controle do fluxo de atendimento.
- **Priorização e categorização:** Possibilitando que os chamados sejam organizados de acordo com a urgência e a natureza do problema.
- **Análise de desempenho:** Permite identificar gargalos e oportunidades de melhoria nos processos de suporte.

Fitzsimmons e Fitzsimmons (2023) destacam que a padronização e a centralização dos processos de atendimento não apenas agilizam a resolução dos chamados, mas também proporcionam uma visão integrada que contribui para a melhoria contínua dos serviços prestados. Assim, sistemas bem estruturados de gerenciamento de *tickets* se tornam essenciais para a fidelização dos clientes e para a construção de uma reputação positiva no mercado.

2.3 APIs e Integração de Sistemas

APIs são componentes fundamentais na arquitetura de sistemas modernos, pois permitem a comunicação entre diferentes plataformas e a troca padronizada de dados. Essa integração é vital para empresas que utilizam múltiplos sistemas para gerenciar suas operações, já que:

- **Facilita a interoperabilidade:** Uma API bem projetada possibilita que diferentes sistemas corporativos se comuniquem de forma eficaz, eliminando barreiras e redundâncias.

- **Promove a flexibilidade:** Ao adotar padrões como REST (*Representational State Transfer* ou Transferência de Estado Representacional) , proposto por Fielding (FIELDING, 2000), as APIs possibilitam a criação de soluções customizáveis que se adaptam às necessidades específicas de cada negócio.

- **Agiliza a inovação:** APIs permitem que novas funcionalidades sejam integradas sem a necessidade de grandes reformulações na infraestrutura existente.

Geewax (2021) destaca que a clareza e a consistência na definição de endpoints, métodos e formatos de dados são essenciais para garantir a escalabilidade e a facilidade de manutenção das integrações. Dessa forma, a utilização de APIs não só otimiza os processos internos, como também abre caminho para a criação de ecossistemas digitais mais robustos e interconectados.

2.4 Segurança e Autenticação

Em um cenário de crescente exposição a ameaças cibernéticas, a segurança dos dados e o controle de acesso são imperativos em qualquer sistema de informação. No contexto do gerenciamento de *tickets*, onde informações sensíveis e dados dos clientes estão envolvidos, adotar mecanismos de segurança robustos é crucial.

Dentre esses mecanismos, a autenticação baseada em *JSON Web Token* (JWT) se destaca por sua eficiência e segurança. O JWT é um padrão aberto para transmissão segura de informações entre partes como um objeto JSON compactado e criptografado.

Basicamente, o processo funciona assim: quando um usuário faz login, o servidor gera um token JWT contendo informações codificadas, como a identidade do usuário e permissões, e o assina digitalmente. Esse token é então enviado ao cliente, que o utiliza para autenticar suas requisições subsequentes ao servidor. O servidor pode validar a autenticidade do usuário sem precisar manter estado, o que melhora a escalabilidade.

As principais vantagens do uso do JWT incluem:

- **Controle de acesso seguro:** Apenas usuários com tokens válidos e autorizados conseguem acessar recursos protegidos, evitando acessos indevidos.

- **Escalabilidade e desempenho:** Por ser *stateless* (não manter estado no servidor), facilita a distribuição do sistema em múltiplos servidores e reduz a necessidade de consultas constantes ao banco de dados para autenticação.

- **Flexibilidade na transmissão de informações:** O token pode conter diversas informações (claims) que ajudam na autorização granular e no controle de permissões.

- **Adoção de boas práticas e compatibilidade:** JWT é amplamente suportado em diversos frameworks, linguagens e plataformas, alinhando-se às recomendações atuais de segurança e protocolos como OAuth 2.0 (CHAPMAN, 2020).

2.5 Comparação com Soluções Existentes

Ferramentas genéricas de gerenciamento de tickets, como Zendesk, Freshdesk e Zoho Desk, têm ganhado ampla adoção no mercado por oferecerem soluções prontas, com funcionalidades voltadas ao atendimento por múltiplos canais, automações básicas e dashboards integrados. De acordo com relatórios de consultorias como a Gartner, essas plataformas se destacam pela facilidade de implantação e escalabilidade em ambientes padronizados. Contudo, análises comparativas disponíveis em repositórios como G2 e Capterra, indicam que essas ferramentas apresentam limitações no que se refere à personalização de fluxos de trabalho e à integração com sistemas corporativos legados ou soluções internas. Empresas com requisitos específicos — como integrações com ERPs próprios, regras de atendimento customizadas ou segmentações por unidade de negócio — relatam dificuldades em adaptar completamente essas plataformas às suas necessidades, o que pode comprometer a eficiência operacional e a aderência aos processos internos.

Característica	**Zendesk**	**Freshdesk**	**Zoho Desk**
Interface amigável e intuitiva	Sim	Sim	Sim
Customização avançada de fluxos e regras	Alta	Baixa	Média
Integração com sistemas legados/ERPs	Limitada	Limitada	Limitada
APIs robustas para integração	Sim	Sim	Sim
Automação de *tickets* e respostas	Sim	Sim	Sim
Gerenciamento de SLA e métricas	Sim	Sim	Sim
Controle de acesso e permissões avançado	Limitado	Limitado	Limitado
Hospedagem local (on-premises)	Não	Não	Não
Preço	Caro	Mais acessível	Mais acessível

Tabela 2.1: Comparativo de Características em Sistemas de Gerenciamento de *Tickets*

Em contrapartida, a proposta deste projeto se diferencia por:

- **Customização dos fluxos de atendimento:** Permite a definição de processos e níveis de prioridade de acordo com as demandas particulares de cada empresa.

- **Flexibilidade na criação de interfaces:** A possibilidade de desenvolver um Front-End personalizado possibilita uma experiência de usuário alinhada à identidade corporativa e às necessidades operacionais.

- **Integração eficaz com sistemas existentes:** Através de uma API bem estruturada, o sistema facilita a comunicação entre diversas plataformas, promovendo a unificação dos processos de atendimento.

Essa abordagem customizada é corroborada por estudos recentes que enfatizam a importância de soluções flexíveis para lidar com a heterogeneidade dos ambientes corporativos, onde a padronização rígida pode limitar a adaptabilidade e a inovação dos processos (FITZSIMMONS; FITZSIMMONS, 2023).

2.6 Desenvolvimento de APIs RESTful

Essa convergência está alinhada às recomendações de (AMUNDSEN, 2022), que destaca como o uso de padrões bem definidos no design de APIs RESTful contribui diretamente para a criação de sistemas mais escaláveis, seguros e adaptáveis às necessidades corporativas. A API desenvolvida não apenas melhora a eficiência dos processos de suporte, mas também oferece:

- **Um ambiente integrado e personalizável:** Adaptável às particularidades de cada organização, promovendo maior controle e eficiência.

- **Facilidade de manutenção e escalabilidade:** Devido à sua arquitetura modular, que permite atualizações e expansões sem interrupções significativas.

- **Melhoria contínua dos serviços:** Por meio do monitoramento sistemático dos atendimentos e da análise de indicadores de desempenho.

CAPÍTULO 3

LEVANTAMENTO E ANÁLISE DE REQUISITOS

Nesta seção, são detalhadas as etapas iniciais para dar início ao desenvolvimento do sistema. O processo foi estruturado em fases que incluem o levantamento de requisitos, análise de requisitos, implementação, testes, correções e entrega. O fluxo de desenvolvimento seguiu um passo a passo lógico, garantindo uma abordagem organizada e eficiente.

3.1 Levantamento e Análise de Requisitos

O levantamento de requisitos foi realizado com base nas necessidades operacionais de um sistema de gerenciamento de *tickets* para otimizar processos de atendimento e suporte. O sistema visa substituir métodos manuais ou pouco integrados, garantindo rastreabilidade, segurança e eficiência na gestão de demandas.

Durante essa análise, foram identificadas diversas funcionalidades essenciais, incluindo o controle eficiente do ciclo de vida dos *tickets*, a definição de prioridades para atendimento e a necessidade de registrar todo o histórico de modificações. Além disso, destacou-se a importância da autenticação segura dos usuários e da aplicação de regras de autorização para garantir que cada perfil tenha acesso apenas às informações pertinentes.

Também foi levantada a necessidade de notificações automáticas via e-mail para manter os usuários informados sobre o andamento dos *tickets*, garantindo um fluxo de comunicação eficiente.

3.1.1 Níveis de acesso do Usuário

Os usuários do sistema são classificados em três categorias, cada uma com permissões específicas:

- **Cliente:** Pode criar *tickets*, visualizar seus próprios chamados, adicionar comentários e anexar arquivos.
- **Funcionário:** Pode visualizar e atender os *tickets* atribuídos a ele, adicionar comentários e registrar o histórico de ações.
- **Gerente:** Possui controle total sobre os *tickets* do seu departamento, podendo criar e gerenciar categorias, definir regras de prioridade e gerar relatórios.

3.1.2 Requisitos Funcionais

Os principais requisitos funcionais do sistema foram definidos conforme a tabela abaixo:

Identificação	**Classificação**	**Ator**	**Objetivo**
Realizar Cadastro	Essencial	Cliente, Funcionário	Permitir que o cliente ou funcionário realize o cadastro no sistema.
Gerenciar Departamento	Essencial	Gerente	Criar, editar e excluir departamentos que atenderão os *tickets*.
Gerenciar Categoria	Essencial	Gerente	Criar, editar ou excluir categorias de *tickets*.
Gerenciar Regra de Prioridade	Essencial	Gerente	Definir critérios para classificação e priorização dos *tickets*.
Gerenciar *Ticket*	Essencial	Cliente, Funcionário, Gerente	Criar, visualizar e modificar *tickets* conforme permissões do perfil.
Registrar Histórico	Essencial	Sistema	Armazenar um registro detalhado das ações realizadas em cada *ticket*.
Realizar Atendimento	Essencial	Funcionário	Atender *tickets* atribuídos e executar as ações necessárias.
Fazer Comentário	Essencial	Cliente, Funcionário	Inserir comentários nos *tickets* para troca de informações.
Anexar Arquivo	Importante	Cliente, Funcionário	Permitir anexação de arquivos em *tickets*.

Gerar Relatório	Importante	Funcionário, Gerente	Criar relatórios sobre os *tickets* e atendimentos realizados.

Tabela 3.1: Tabela de requisitos funcionais

3.1.3 Regras de Negócio

O sistema de gerenciamento de *tickets* foi desenvolvido para oferecer controle eficiente sobre os atendimentos realizados pelas empresas. As regras de negócio estabelecidas garantem o funcionamento adequado das funcionalidades, respeitando os níveis de acesso dos usuários e a lógica dos fluxos de atendimento.

1. Autenticação e Autorização

- O sistema utiliza autenticação via JWT, garantindo segurança na troca de informações.
- O token contém dados do usuário, incluindo seu ID e perfil de acesso (`ROLE`), sendo validado a cada requisição.

2. Regras de Acesso aos *Tickets*

- **Clientes:** Apenas podem acessar os *tickets* que abriram.
- **Funcionários:** Somente podem visualizar e alterar os *tickets* atribuídos a eles.
- **Gerentes:** Têm acesso a todos os *tickets* do seu departamento.

3. Fluxo de Trabalho dos *Tickets*

- **Criação:**
  - No momento da criação, o sistema classifica a prioridade do *ticket*.
  - É enviada uma notificação automática ao cliente e ao responsável pelo atendimento.
  - O tempo máximo de resolução é calculado com base na prioridade e na categoria do chamado.

- **Atualização:**
  - Todas as mudanças nos *tickets* são registradas no histórico.
  - O sistema verifica e registra alterações feitas nos campos do *ticket*.
  - Caso o status de um *ticket* seja alterado para "EM ANDAMENTO", o cliente recebe uma notificação automática.
- **Comentários:**
  - Apenas usuários autorizados podem adicionar comentários.
  - Clientes só podem comentar nos seus próprios *tickets*.
  - Funcionários só podem comentar nos *tickets* pelos quais são responsáveis.
  - Gerentes só podem comentar nos *tickets* do seu departamento.
- **Criação de Usuário:**
  - O sistema valida os dados informados no cadastro, incluindo CPF e e-mail.
  - Não é permitido o cadastro de usuários duplicados.
  - A senha do usuário é encriptada antes de ser armazenada.
  - Após o cadastro, um e-mail de confirmação é enviado ao usuário.

3.1.4 Envio de E-mails

O sistema possui um serviço de envio de e-mails automáticos para manter os usuários informados sobre eventos relevantes. As principais notificações incluem:

- **Cadastro de usuário:** Confirmação da criação da conta e instruções para ativação.
- **Criação de *ticket*:** Notificação ao cliente e ao responsável pelo atendimento.
- **Atualização de status:** Aviso ao cliente sempre que o *ticket* sofrer alterações.

3.1.5 Diagrama de Casos de Uso

A Figura 3.1 ilustra as interações entre os usuários e o sistema, evidenciando as principais funcionalidades acessíveis a cada perfil. Esse diagrama fornece uma visão geral do comportamento do sistema a partir da perspectiva dos usuários, facilitando a compreensão dos requisitos e do escopo das operações disponíveis.

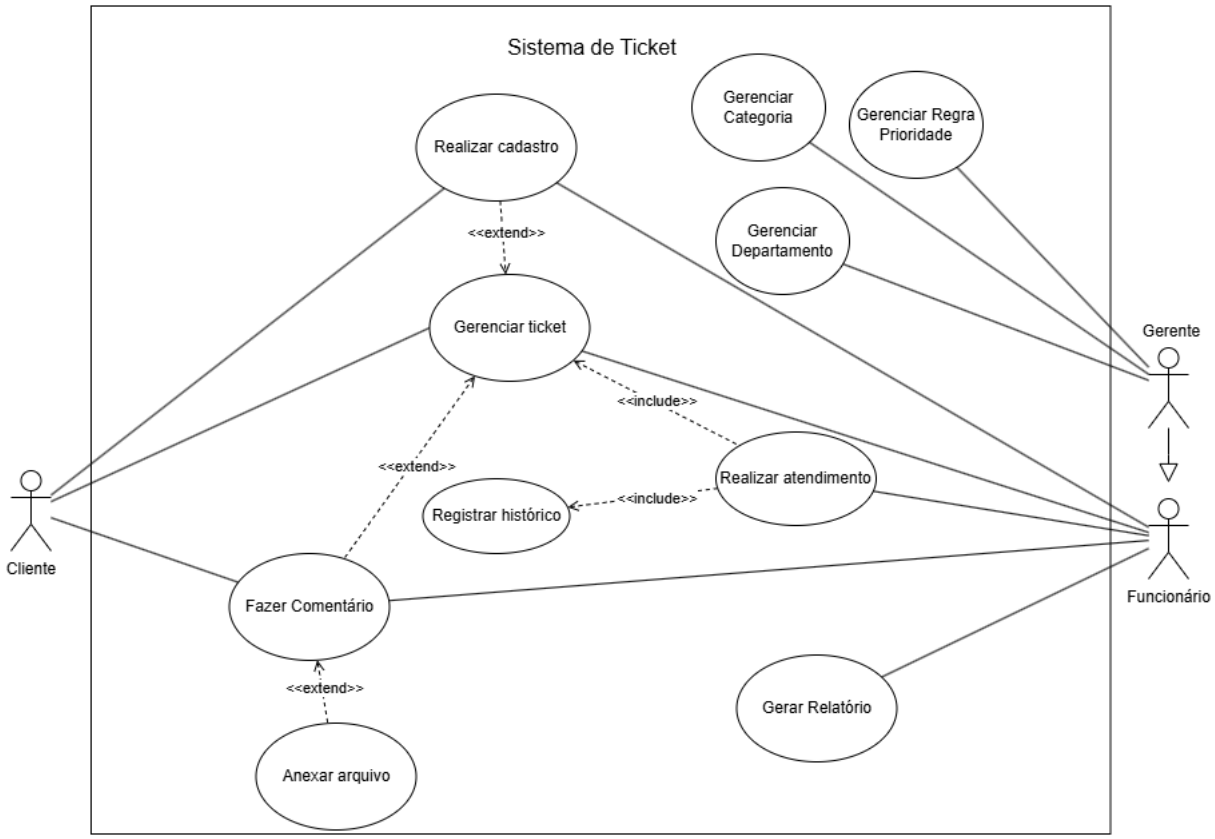


Figura 3.1: Diagrama de casos de uso

UC01 - Fazer Cadastro

Antes de apresentar os casos de uso do sistema, é importante contextualizar que eles representam as interações entre os usuários (atores) e o sistema, descrevendo o comportamento esperado em diferentes situações. Cada caso de uso é descrito de forma estruturada, utilizando um modelo tabular que contempla os principais elementos, como atores envolvidos, pré e pós-condições, ações e restrições.

A Tabela 3.2 apresenta o caso de uso UC01, que descreve o processo de cadastro de um novo usuário no sistema:

Tabela 3.2: Descrição do Caso de Uso UC01 - Fazer Cadastro

Nome do caso de uso	UC01 - Fazer Cadastro
Ator principal	Usuário
Atores secundários	-
Resumo	Permite que um paciente faça o cadastro no sistema.
Pré-condições	-
Pós-condições	Usuário cadastrado recebe e-mail de confirmação.
Cenário Principal	
Ações do ator	Ações do sistema
1) Preencher dados pessoais (nome, e-mail, senha)	2) Validar informações
3) Confirmar cadastro	4) Registrar usuário no sistema
5) Enviar e-mail de confirmação	
Restrições/Validações	- E-mail deve ser único e válido - Todos os campos obrigatórios devem ser preenchidos

UC02 - Gerenciar *Ticket*

A Tabela 3.3 apresenta o caso de uso UC02, que descreve as ações relacionadas à criação, edição e visualização de *tickets* por diferentes perfis de usuário no sistema, respeitando os níveis de acesso definidos.

Tabela 3.3: Descrição do Caso de Uso UC02 - Gerenciar *Ticket*

Nome do caso de uso	UC02 - Gerenciar *Ticket*
Ator principal	Cliente, Funcionário, Gerente
Atores secundários	-
Resumo	Permite criar, alterar e visualizar *tickets* no sistema com diferentes níveis de acesso.
Pré-condições	Usuário deve estar autenticado no sistema.
Pós-condições	Registro de *ticket* atualizado no histórico do sistema.
Cenário Principal	
Ações do ator	Ações do sistema
1) Preencher dados do *ticket* (título, descrição, categoria)	2) Validar informações 3) Registrar *ticket* no sistema
4) Buscar *ticket* existente	5) Exibir detalhes completos
6) Modificar informações (status/descrição)	7) Atualizar registro
Restrições/Validações	- Histórico de alterações deve ser registrado - Clientes só podem criar/visualizar próprios *tickets* - Gerentes têm acesso total a todos os *tickets* do seu departamento - Funcionários podem editar *tickets* atribuídos a eles

UC03 - Fazer Comentário

A Tabela 3.4 apresenta o caso de uso UC03, que descreve o processo de inclusão de comentários em *tickets* já existentes. Essa funcionalidade é essencial para promover a comunicação entre usuários do sistema, permitindo que clientes e funcionários acompanhem o andamento e interajam nos chamados.

Tabela 3.4: Descrição do Caso de Uso UC03 - Fazer Comentário

Nome do caso de uso	UC03 - Fazer Comentário
Ator principal	Cliente, Funcionário
Atores secundários	-
Resumo	Permite adicionar comentários aos *tickets* para comunicação entre as partes.
Pré-condições	- *Ticket* deve existir no sistema - Usuário deve ter acesso ao *ticket*
Pós-condições	Comentário registrado com data/hora no histórico do *ticket*.
Cenário Principal	
Ações do Ator	Ações do sistema
1) Preencher dados do comentário (título, descrição)	2) Validar informações e se o *ticket* do comentário existe 3) Registrar comentário no sistema 4) Atualizar histórico do *ticket*
Restrições/Validações	- Cliente só comenta em próprios *tickets* - Funcionário só comenta em *tickets* atribuídos

3.1.6 Diagrama de Sequência

O diagrama de sequência é um artefato da UML que descreve a interação entre objetos ou componentes do sistema ao longo do tempo. Ele representa, em uma visão temporal, a troca de

mensagens entre atores (Cliente, Controller, Service, Repositório) e evidencia a ordem e o fluxo das chamadas, desde a requisição até a resposta final. Esse diagrama é especialmente útil para:

- Validar cenários de uso complexos, mostrando passo a passo como as operações são encadeadas.
- Identificar pontos de integração entre camadas (por exemplo, controller–service–repository).
- Documentar o comportamento dinâmico do sistema, facilitando a comunicação entre desenvolvedores e analistas.

A Figura 3.2 demonstra o fluxo de criação do usuário, na qual, ao receber a requisição de criação de usuário, o *Controller* delega a operação ao *Service*. Este primeiro verifica se já existe um usuário com o mesmo e-mail ou CPF; em caso positivo, retorna um erro de duplicação (400 Bad Request). Se os dados forem válidos, o *Service* codifica a senha (BCrypt), persiste a entidade no banco e dispara um e-mail de confirmação. Finalmente, retorna ao *Controller* a resposta de sucesso (201 Created), que a envia de volta ao cliente.

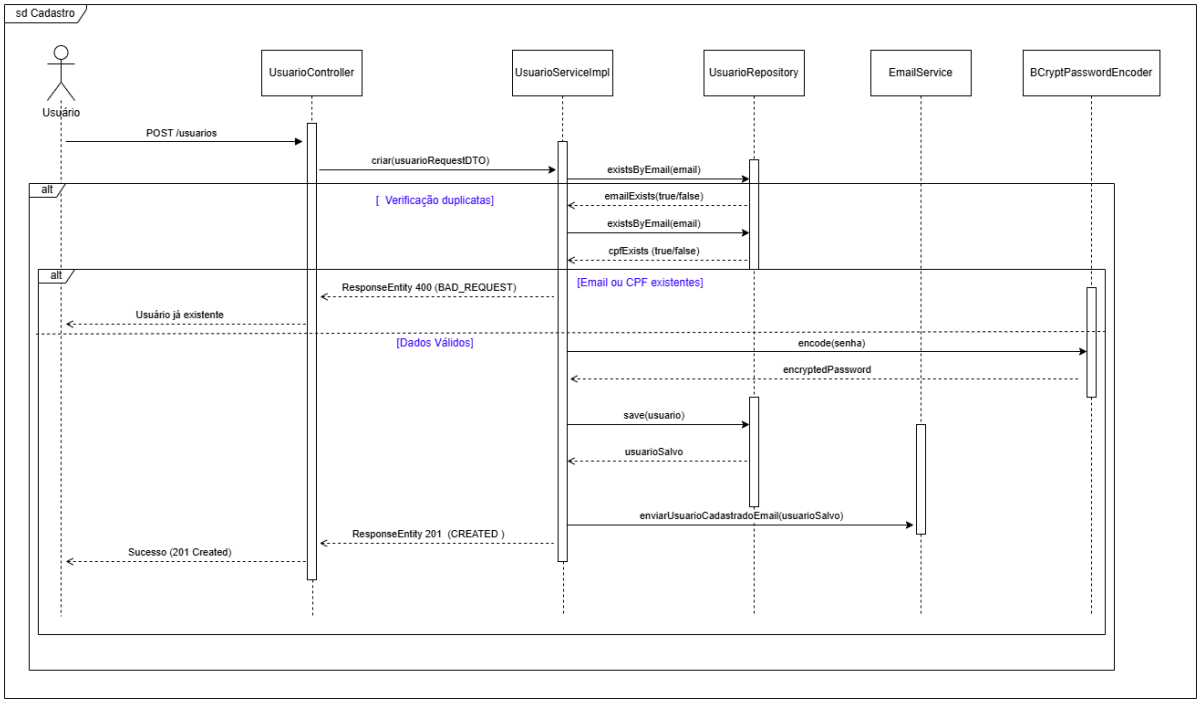


Figura 3.2: Diagrama de sequência do usuário

Já a Figura 3.3 ilustra o diagrama de sequência para o fluxo de criação de uma nova regra de prioridade. Esse diagrama evidencia a interação temporal entre os componentes do sistema: o `Cliente`, o `RegraPrioridadeController`, o serviço `RegraPrioridadeServiceImpl`, os serviços de `Categoria` e `Departamento` e o `RegraPrioridadeRepository`.

A Figura 3.3 ilustra, passo a passo, o processo de criação de uma nova regra de prioridade a partir de uma requisição feita por um cliente.

O fluxo se inicia quando o cliente envia os dados da nova regra ao *Controller*. Em seguida, o *Controller* repassa essas informações ao *Service*, onde ocorre o processamento da lógica de negócio. Dentro do *Service*, diversas etapas importantes são executadas:

- Os dados recebidos são convertidos para o formato interno utilizado pelo sistema;
- O sistema verifica se a categoria e o departamento informados existem, consultando serviços especializados;
- Todos os dados são validados e preparados para persistência.

Após essas verificações, a nova regra é salva no banco de dados. Como medida de otimização, o sistema limpa o cache de regras, garantindo que as próximas consultas retornem informações atualizadas. Por fim, o sistema formata a resposta e a envia de volta ao cliente, confirmando a criação da regra com sucesso (código HTTP 201).

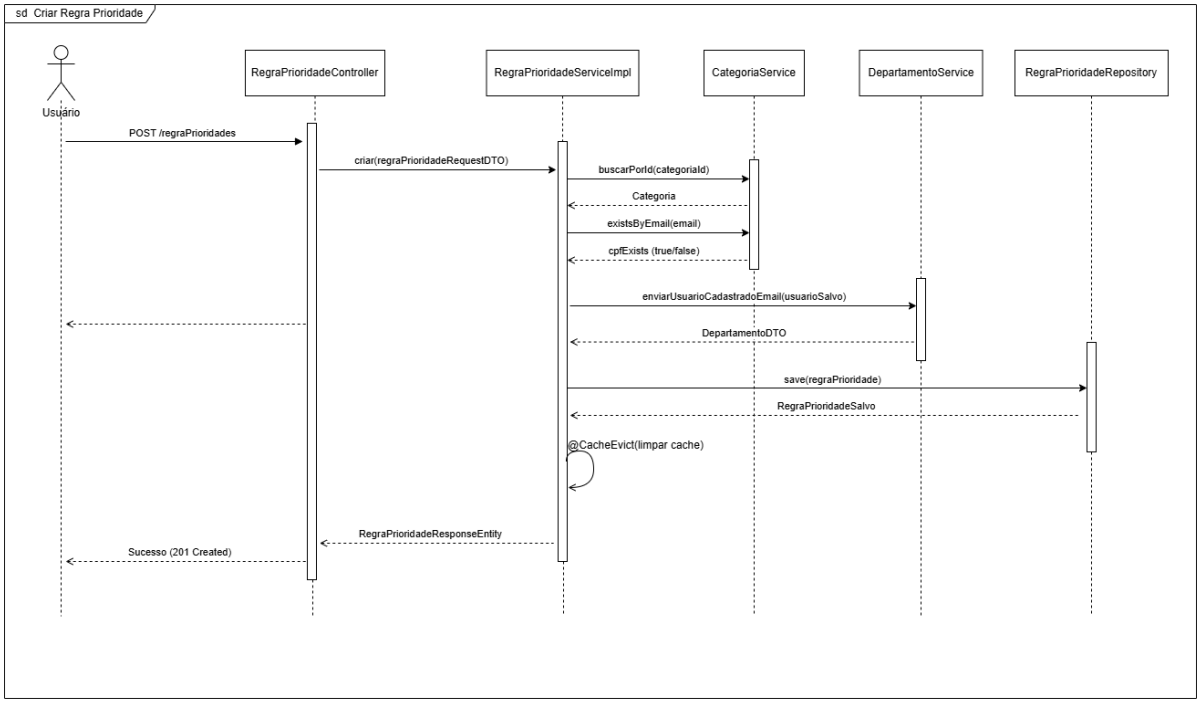


Figura 3.3: Diagrama de sequência da Regra de Prioridade

3.1.7 Diagrama de Classes de domínio

A Figura 3.4 apresenta o diagrama de classes que representa a estrutura fundamental de um sistema de gerenciamento de *tickets*, típico de aplicações como helpdesk ou suporte técnico. Abaixo, são detalhados os principais componentes e seus relacionamentos:

Principais Classes

- ***Ticket*:** Classe central do sistema. Armazena informações essenciais como descrição, datas (criação, modificação, prazo), status atual e associações com outras entidades.
- **Usuario:** Representa os usuários cadastrados no sistema. Contém atributos como nome, e-mail, status e tipo de usuário, definido por um enumerador (`UsuarioRole`).
- **Departamento:** Modela os setores da organização. Inclui dados como nome, informações de contato e horário de funcionamento.
- **Comentario:** Permite que os usuários adicionem observações ou mensagens relacionadas aos *tickets*.
- **TicketHistorico:** Armazena registros das alterações realizadas nos *tickets*, funcionando como um log de mudanças.

- **ReparPrioridade:** Responsável pela gestão das prioridades dos *tickets*, com métodos específicos para cálculo do nível de prioridade e do tempo restante.
- **Anexo:** Permite a associação de arquivos aos *tickets* ou a outras entidades do sistema.

Tipos Enumerados (Enums)

- **StatusTicket:** Define os estados possíveis de um *ticket*, como `Aberto`, `Em_Andamento` e `Finalizado`.
- **UsuarioRole:** Enumeração que representa os papéis dos usuários no sistema, como `Administrador`, `Funcionário` e `Cliente`.

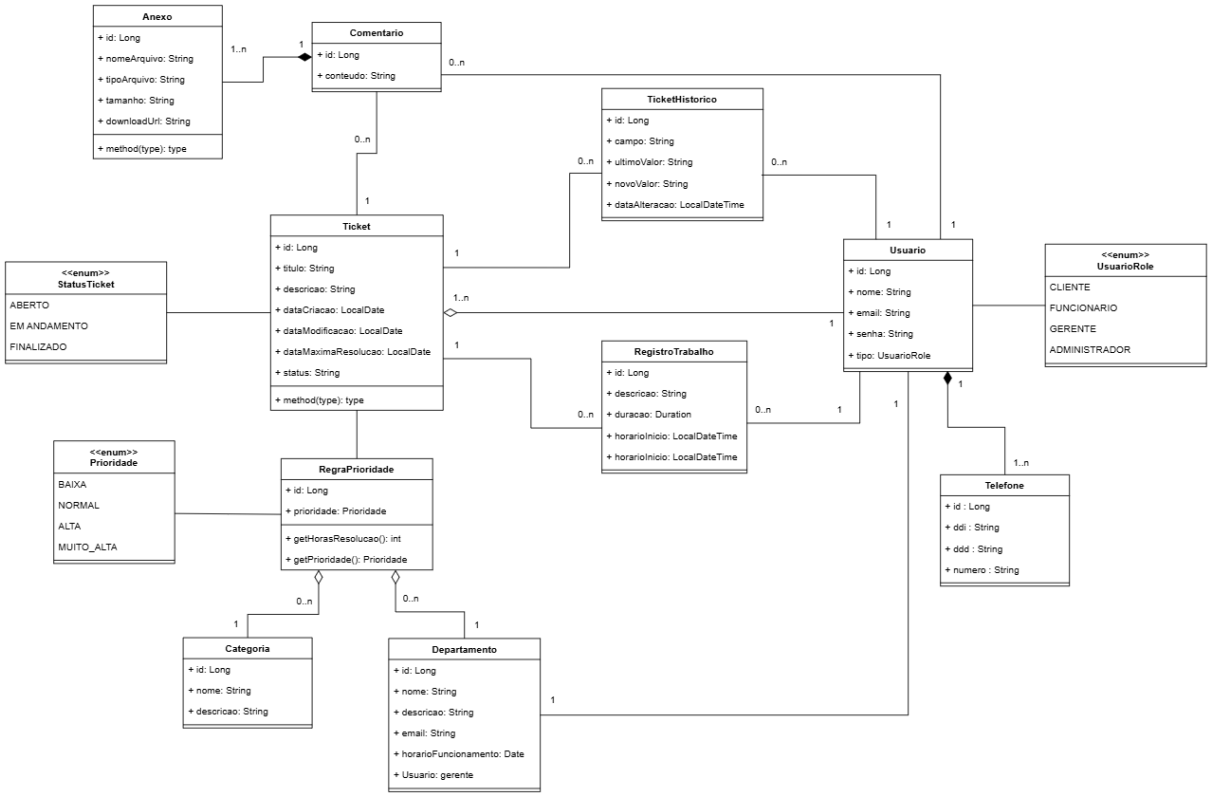


Figura 3.4: Diagrama de classe

3.2 Definição de Tecnologias e Ferramentas

Para o desenvolvimento do sistema, foram definidas as seguintes tecnologias, considerando critérios como robustez, compatibilidade com o escopo do projeto, comunidade ativa e facilidade de integração entre os componentes:

Tecnologia	Aspecto	Descrição
Java	Linguagem	Desenvolvimento do Back-end (API REST)
Spring Boot	Framework	Framework para aplicações Java
RabbitMQ	Mensageria	Fila para comunicação assíncrona
MySQL	SGBD	Banco de dados relacional
IntelliJ IDEA	IDE	Desenvolvimento do Back-end
Git	Versionamento de Código	Controle de versão local
GitHub	Versionamento de Código	Armazenamento e colaboração em nuvem
Postman	Testes de API	Testes e validação das requisições da API

Tabela 3.5: Tabela consolidada de tecnologias, aspectos e descrições

CAPÍTULO 4

IMPLEMENTAÇÃO DO BACK-END (API REST)

Após o levantamento e análise dos requisitos, foi inicializado o desenvolvimento do sistema, sendo feito sua construção na IDE IntelliJ Idea, utilizando a linguagem Java e o Framework Spring.

4.1 Configuração Inicial

A primeira etapa do desenvolvimento do sistema foi a criação do projeto Spring Boot, utilizando a IDE IntelliJ IDEA. Para garantir a integração de todos os componentes necessários, foram adicionadas as seguintes dependências ao arquivo pom.xml:

Dependência	Descrição
Spring Boot Starter Data JPA	Para integração com o banco de dados e utilização do JPA para persistência de dados.
Spring Boot Starter Web	Habilita o desenvolvimento de APIs RESTful no sistema.
MySQL Connector	Conecta o sistema ao banco de dados MySQL em ambiente de produção.
Bucket4j	Utilizado para implementar controle de limites de requisições (rate limiting) na API.
Springdoc OpenAPI	Para documentação automática da API no formato OpenAPI.
Ehcache	Implementação de cache para melhorar a performance do sistema.
Lombok	Reduz a verbosidade do código, utilizando anotações como @Getter, @Setter, e @Builder.
Spring Boot Starter AMQP	Integração com filas AMQP, como RabbitMQ, para gerenciamento de mensagens assíncronas.
Spring Boot Starter Validation	Validação de dados nas requisições HTTP.
Flyway	Controle de versionamento do banco de dados, realizando migrações e manutenção do esquema.
Spring Security	Implementação de segurança na aplicação, incluindo autenticação e autorização via JWT.
Spring Security Test	Para escrever testes de segurança e garantir que a aplicação esteja protegida.

Tabela 4.1: Tabela de dependências utilizadas

4.2 Projeto do Banco de Dados

O banco de dados da aplicação de gerenciamento de *tickets* foi projetado para armazenar e organizar informações essenciais sobre os chamados e suas interações. A modelagem das tabelas foi baseada nas entidades principais do sistema, como Usuário, *Ticket* e Registro de Trabalho, garantindo que os relacionamentos entre elas sejam bem definidos e otimizados para consultas eficientes.

A geração das tabelas, colunas e chaves estrangeiras foi realizada automaticamente pelo Spring Data JPA, a partir das classes de entidade (POJOs) anotadas com as especificações do JPA. Essa abordagem facilita a manutenção do esquema do banco de dados, assegurando que as estruturas estejam sempre alinhadas com o modelo de domínio da aplicação.

Essa modelagem permite a escalabilidade do sistema e a integridade dos dados, garantindo que todas as operações realizadas pelos usuários sejam devidamente registradas e acessíveis conforme as permissões definidas.

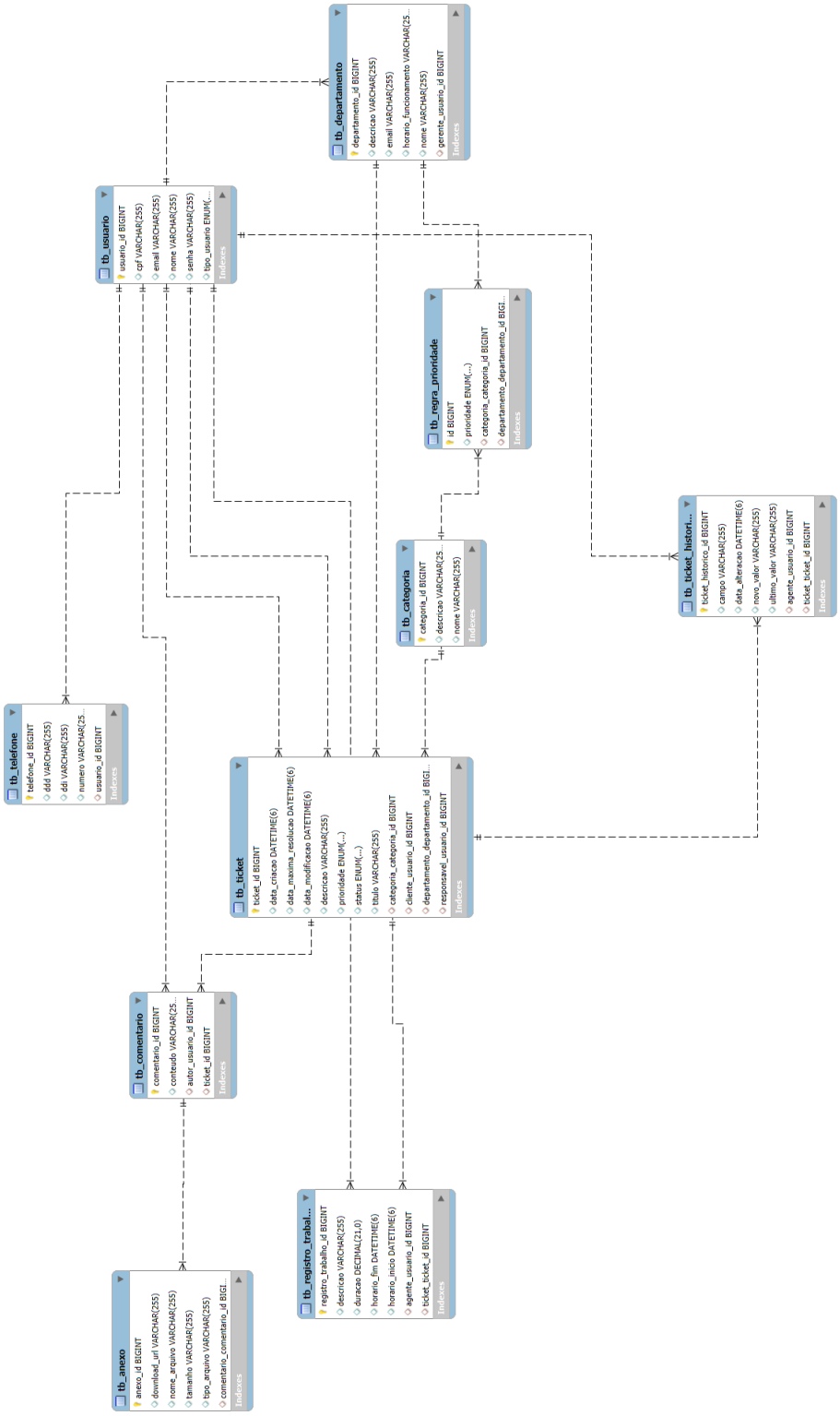
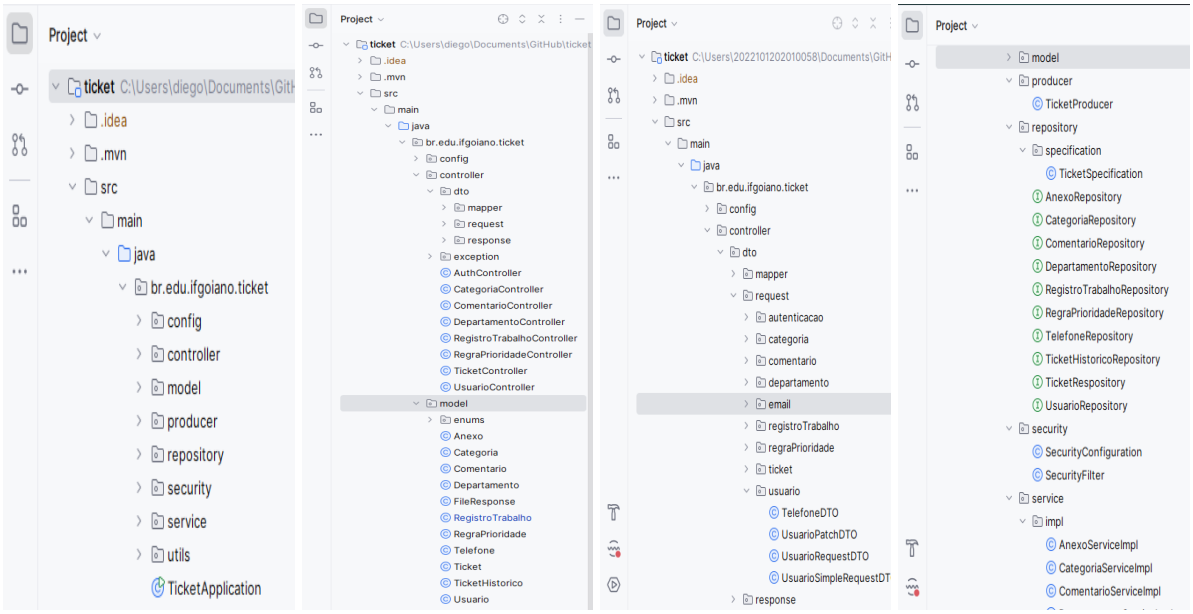


Figura 4.1: Diagrama de classes do banco de dados

4.3 Estrutura de Pacotes do Back-end

A Figura 4.2 mostra a estrutura de pacotes do back-end em que se observa a estutura de pacotes geral e outras mais específicas. A Figura 4.2(a), mostra os pacotes da aplicação. A

Figura 4.2(b), mostra os subpacotes do controller e model. A Figura 4.2(c), detalha sobre o subpacote de dto. Por fim, a Figura 4.2(d), detalha o subpacote de repository e service.



(a) Estrutura geral. **(b)** Estrutura subpacotes 1. **(c)** Estrutura subpacotes 2. **(d)** Estrutura subpacotes 3.

Figura 4.2: Estrutura de pacotes do Back-end

4.4 Endpoints da api

Nesta seção, são apresentados os endpoints da API do sistema de gerenciamento de *tickets*. A API oferece funcionalidades para autenticação, gestão de usuários, categorias, departamentos, regras de prioridade, *tickets*, registros de trabalho e comentários. Abaixo estão os detalhes de cada conjunto de endpoints.

4.4.1 Endpoints de Autenticação

A autenticação é um mecanismo essencial para assegurar que apenas usuários devidamente autorizados tenham acesso aos recursos do sistema. Os principais endpoints relacionados ao processo de autenticação estão descritos na Tabela 4.2.

Tabela 4.2: Endpoints da API de Autenticação

Método	Endpoint	Descrição
POST	`/api/v1/auth/login`	Endpoint utilizado para autenticação de usuários. Recebe um JSON contendo credenciais e retorna um token JWT para acesso autenticado.
POST	`/api/v1/auth/refresh`	Permite renovar o token de autenticação quando este expira, retornando um novo token JWT.

4.4.2 Endpoints de Usuários

Os endpoints relacionados aos usuários possibilitam a realização das operações de criação, consulta, atualização e remoção de registros no sistema. Tais funcionalidades são essenciais para o gerenciamento de acessos e o controle de permissões dentro do ambiente da aplicação, garantindo a integridade e a segurança das interações realizadas. A Tabela 4.3 apresenta os principais endpoints da API responsáveis por essas operações.

Tabela 4.3: Endpoints da API de Usuários

Método	Endpoint	Descrição
POST	`/api/v1/usuarios`	Criação de um novo usuário no sistema.
GET	`/api/v1/usuarios`	Retorna a lista de todos os usuários cadastrados.
GET	`/api/v1/usuarios/{id}`	Retorna um usuário específico com base no ID.
PUT	`/api/v1/usuarios/{id}`	Atualiza as informações de um usuário específico.
DELETE	`/api/v1/usuarios/{id}`	Remove um usuário do sistema com base no ID.

4.4.3 Endpoints de Categorias

A gestão de categorias desempenha um papel fundamental na organização dos *tickets*, permitindo que sejam classificados em grupos específicos. Essa classificação contribui para uma melhor estruturação, priorização e tratamento das solicitações. Os principais endpoints responsáveis pelas operações de criação, consulta, atualização e remoção de categorias estão descritos na Tabela 4.4.

Tabela 4.4: Endpoints da API de Categorias

Método	Endpoint	Descrição
POST	`/api/v1/categorias`	Criação de uma nova categoria no sistema.
GET	`/api/v1/categorias`	Retorna a lista de todas as categorias cadastradas.
GET	`/api/v1/categorias/{id}`	Retorna uma categoria específica com base no ID.
PUT	`/api/v1/categorias/{id}`	Atualiza as informações de uma categoria específica.
DELETE	`/api/v1/categorias/{id}`	Remove uma categoria do sistema com base no ID.

4.4.4 Endpoints de Departamentos

Os departamentos são utilizados para organizar os *tickets* com base nas áreas ou equipes responsáveis, contribuindo para uma melhor distribuição das demandas e agilidade no atendimento. A Tabela 4.5 apresenta os principais endpoints da API responsáveis pela gestão dos departamentos no sistema.

Tabela 4.5: Endpoints da API de Departamentos

Método	Endpoint	Descrição
POST	`/api/v1/departamentos`	Criação de um novo departamento no sistema.
GET	`/api/v1/departamentos`	Retorna a lista de todos os departamentos cadastrados.
GET	`/api/v1/departamentos/{id}`	Retorna um departamento específico com base no ID.
PUT	`/api/v1/departamentos/{id}`	Atualiza as informações de um departamento específico.
DELETE	`/api/v1/departamentos/{id}`	Remove um departamento do sistema com base no ID.

4.4.5 Endpoints de Regras de Prioridade

As regras de prioridade são utilizadas para definir o nível de urgência no tratamento dos *tickets*, permitindo que o sistema atribua diferentes graus de atenção conforme critérios previamente estabelecidos. A Tabela 4.6 apresenta os principais endpoints da API responsáveis pela gestão dessas regras.

Tabela 4.6: Endpoints da API de Regras de Prioridade

Método	Endpoint	Descrição
POST	`/api/v1/regraPrioridades`	Criação de uma nova regra de prioridade no sistema.
GET	`/api/v1/regraPrioridades`	Retorna a lista de todas as regras de prioridade cadastradas.
PUT	`/api/v1/regraPrioridades/{id}`	Atualiza as informações de uma regra de prioridade específica com base no ID.
DELETE	`/api/v1/regraPrioridades/{id}`	Remove uma regra de prioridade do sistema com base no ID.

4.4.6 Endpoints de *Tickets*

Os *tickets* representam os problemas, dúvidas ou solicitações registradas pelos usuários e que demandam algum tipo de resolução por parte da equipe responsável. Eles constituem o elemento central do sistema de atendimento, permitindo o acompanhamento de todo o ciclo de vida de uma demanda. A Tabela 4.7 apresenta os principais endpoints da API utilizados na gestão dos *tickets*, incluindo funcionalidades de criação, consulta, atualização e remoção.

Tabela 4.7: Endpoints da API de *Tickets*

Método	Endpoint	Descrição
POST	`/api/v1/tickets`	Criação de um novo *ticket* no sistema.
GET	`/api/v1/tickets`	Retorna a lista de todos os *tickets* com filtros opcionais (titulo, status, prioridade, responsável, data de início e data de fim).
GET	`/api/v1/tickets/{id}`	Retorna os detalhes de um *ticket* específico com base no ID.
PUT	`/api/v1/tickets/{id}`	Atualiza as informações de um *ticket* específico com base no ID.
DELETE	`/api/v1/tickets/{id}`	Remove um *ticket* do sistema com base no ID.

4.4.7 Endpoints de Registros de Trabalho

Os registros de trabalho estão diretamente vinculados ao acompanhamento do progresso na resolução dos *tickets*, permitindo que sejam documentadas as ações executadas ao longo do atendimento. Esses registros contribuem para a rastreabilidade, análise de desempenho e melhoria contínua dos processos de suporte. A Tabela 4.8 apresenta os principais endpoints da API utilizados na gestão dos registros de trabalho.

Tabela 4.8: Endpoints da API de Registros de Trabalho

Método	Endpoint	Descrição
POST	`/api/v1/registroTrabalho`	Criação de um novo registro de trabalho associado a um *ticket*.
GET	`/api/v1/registroTrabalho`	Retorna todos os registros de trabalho associados a um *ticket* específico.
PUT	`/api/v1/registroTrabalho/{id}`	Atualiza um registro de trabalho específico com base no ID fornecido.
DELETE	`/api/v1/registroTrabalho/{id}`	Remove um registro de trabalho específico com base no ID fornecido.

4.4.8 Endpoints de Comentários

Os comentários são utilizados para registrar observações, atualizações ou interações realizadas durante o ciclo de vida de um *ticket*, contribuindo para a comunicação entre os envolvidos no atendimento. Eles também podem conter anexos, como imagens ou documentos que auxiliem na resolução da demanda. A Tabela 4.9 apresenta os principais endpoints da API responsáveis pela gestão dos comentários no sistema.

Tabela 4.9: Endpoints da API de Comentários

Método	Endpoint	Descrição
POST	`/api/v1/comentarios`	Criação de um novo comentário associado a um *ticket*, com possibilidade de incluir anexo.
GET	`/api/v1/comentarios`	Retorna todos os comentários associados a um *ticket* específico.
PUT	`/api/v1/comentarios/{id}`	Atualiza um comentário específico com base no ID fornecido.
DELETE	`/api/v1/comentarios/{id}`	Remove um comentário específico com base no ID fornecido.
DELETE	`/api/v1/comentarios/{id}/anexo`	Deleta um anexo de um comentário especificado pelo ID e nome do arquivo.

4.5 Trechos de Código

O sistema possui diversas regras de negócio que garantem o correto funcionamento das operações. A seguir, são apresentados trechos de código que exemplificam algumas dessas regras, aplicadas em diferentes funcionalidades, como cadastro, atualização, exclusão e consulta de dados

4.5.1 Usuário

O Listing 4.1 apresenta a implementação das regras de negócio para a criação e consulta de usuários. No primeiro trecho de código, o método criar recebe um objeto UsuarioRequestDTO, converte os dados para a entidade Usuario e valida a existência de e-mail e CPF antes de persistir no banco de dados. Caso já exista um usuário com os mesmos dados, a operação é interrompida com uma resposta de erro. Se os dados forem válidos, a senha do usuário é criptografada e o registro é salvo, seguido do envio de um e-mail de confirmação.

No segundo trecho, o método buscaPorId verifica a identidade e as permissões do usuário autenticado antes de permitir o acesso aos dados de um usuário específico. Se um cliente tentar acessar informações de outro usuário, a requisição é negada. Caso contrário, o usuário é recuperado do banco de dados e retornado como um UsuarioResponseDTO.

```
@Service
public class UsuarioServiceImpl implements UsuarioService,
    UserDetailsService {

    @Override
    public ResponseEntity<MessageResponseDTO> criar(UsuarioRequestDTO
        usuarioCreate) {
        Usuario usuario = mapper.mapTo(usuarioCreate, Usuario.class);
        usuario.setContatos(mapper.toList(usuarioCreate.getContatos()
            , Telefone.class));
        usuario.setTipoUsuario(UsuarioRole.CLIENTE);
        usuario.getContatos().forEach(telefone -> telefone.setUsuario
            (usuario));

        boolean emailExists = usuarioRepository.existsByEmail(
            usuarioCreate.getEmail().toLowerCase());
        boolean cpfExists = usuarioRepository.existsByCpf(
            usuarioCreate.getCpf().trim());

        if(emailExists || cpfExists) {
            String errorMessage = "Usuario ja cadastrado. Campo
                duplicado: ";
            errorMessage += emailExists ? "Email" : "CPF";
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body
                (MessageResponseDTO
                    .builder()
                    .code(400)
                    .status("Bad Request")
                    .message(errorMessage)
                    .build());
        }

        String encryptedPassword = new BCryptPasswordEncoder().encode
```

```
            (usuario.getSenha());
26          usuario.setSenha(encryptedPassword);
27
28          var usuarioSalvo = usuarioRepository.save(usuario);
29
30          emailService.enviarUsuarioCadastradoEmail(usuarioSalvo);
31          return ResponseEntity.status(HttpStatus.CREATED).body(
               MessageResponseDTO
32                  .builder()
33                  .code(201)
34                  .status("Created")
35                  .message("Usuario criado com sucesso.")
36                  .build());
37      }
38
39      @Override
40      public UsuarioResponseDTO buscaPorId(Long uuid) {
41          Long uuidAuth = Long.valueOf((String) SecurityContextHolder.
               getContext().getAuthentication().getPrincipal());
42
43          boolean somenteCliente = SecurityContextHolder.getContext().
               getAuthentication()
44                  .getAuthorities().stream()
45                  .map(GrantedAuthority::getAuthority)
46                  .allMatch(role -> role.equals("ROLE_CLIENTE"));
47
48          if(somenteCliente && !Objects.equals(uuidAuth, uuid))
49              throw new CustomAccessDeniedException("Acesso negado.Voce
                   nao tem permissao para acessar este recurso.");
50
51          Usuario usuario = usuarioRepository.findById(uuid)
52                  .orElseThrow(() -> new ResourceNotFoundException("Nao
                       foi encontrado nenhum usuario com esse id."));
53          return mapper.mapTo(usuario, UsuarioResponseDTO.class);
54      }
55  }
```

Listing 4.1: Regras do Usuário da API

4.5.2 Regra Prioridade

O Listing 4.2 apresenta a implementação das regras de negócio para a criação de regras de prioridade no sistema. O método criar recebe um objeto RegraPrioridadeRequestDTO e o converte para a entidade RegraPrioridade. Em seguida, busca a categoria e o departamento associados à regra, garantindo que ambos existam antes de prosseguir com a persistência dos dados.

Além disso, a anotação @CacheEvict é utilizada para limpar o cache de regras de

prioridade sempre que uma nova regra for criada, garantindo que os dados armazenados estejam sempre atualizados. Por fim, a regra de prioridade é salva no banco de dados e retornada como um RegraPrioridadeResponseDTO.

```
@Service
public class RegraPrioridadeServiceImpl implements
   RegraPrioridadeService {

    @Override
    @CacheEvict(value = "regraPrioridadeCache", allEntries = true)
    public RegraPrioridadeResponseDTO criar(RegraPrioridadeRequestDTO
        regraPrioridadeRequestDTO) {
        RegraPrioridade regraPrioridade = mapper.mapTo(
           regraPrioridadeRequestDTO, RegraPrioridade.class);
        Categoria categoria = categoriaService.buscaPorId(
           regraPrioridade.getCategoria().getId());
        Departamento departamento = mapper.mapTo(departamentoService.
           buscarPorId(regraPrioridade.getDepartamento().getId()),
           Departamento.class);
        regraPrioridade.setCategoria(categoria);
        regraPrioridade.setDepartamento(departamento);
        return mapper.mapTo(regraPrioridadeRepository.save(
           regraPrioridade), RegraPrioridadeResponseDTO.class);
    }
}
```

Listing 4.2: Regras de Prioridade

4.5.3 Departamento

O Listing 4.3 apresenta a implementação das regras de negócio para a criação de um departamento no sistema. O método criar recebe um objeto DepartamentoRequestDTO e o converte para a entidade Departamento.

Antes de persistir os dados, o sistema valida se o usuário informado como gerente realmente possui esse cargo. Para isso, busca os detalhes do usuário pelo ID e verifica se ele é um gerente. Caso contrário, uma exceção ResourceNotFoundException é lançada, impedindo o cadastro com informações inválidas.

Após essa validação, o usuário é associado ao departamento, e a entidade é salva no banco de dados, retornando um DepartamentoResponseDTO.

```
@Service
public class DepartamentoServiceImpl implements DepartamentoService {

    @Override
    @CacheEvict(value = "departamentoCache", allEntries = true)
    public DepartamentoResponseDTO criar(DepartamentoRequestDTO
        departamentoRequestDTO) {
         Departamento departamento = mapper.mapTo(
            departamentoRequestDTO, Departamento.class);
         UsuarioResponseDTO usuarioResponseDTO = usuarioService.
            buscaPorId(departamento.getGerente().getId());
         if(usuarioService.verificarSeUsuarioEhGerente(
            usuarioResponseDTO.getId()))
             throw new ResourceNotFoundException("Usuario enviado nao
                e um gerente.");

         Usuario usuario = mapper.mapTo(usuarioResponseDTO,Usuario.
            class);
         departamento.setGerente(usuario);
         return mapper.mapTo(departamentoRepository.save(departamento)
            , DepartamentoResponseDTO.class);
    }
}
```

Listing 4.3: Departamento

4.5.4 *Ticket*

O Listing 4.4 apresenta a implementação do serviço responsável pelo gerenciamento de *tickets* no sistema. A classe TicketServiceImpl contém métodos para criação, busca, atualização e remoção de *tickets*, além de funcionalidades como envio de e-mails e geração de relatórios em CSV.

O método criar recebe um TicketRequestDTO e realiza diversas operações antes de persistir o *ticket* no banco de dados. Primeiro, busca e valida as entidades associadas, como Categoria, Departamento e RegraPrioridade. Em seguida, identifica o cliente e o responsável pelo *ticket*, atribui a prioridade com base nas regras definidas e calcula a data máxima para resolução. Após salvar a entidade, um e-mail de notificação é enviado.

O serviço também implementa filtros para listar *tickets* conforme o perfil do usuário, garantindo que clientes, funcionários e gerentes vejam apenas os chamados pertinentes ao seu papel. Além disso, o método atualizar registra um histórico de alterações nos campos modificados, preservando um log das mudanças realizadas.

Por fim, a classe contém métodos para alterar o status de um *ticket* para "Em Andamento"e remover *tickets*.

```
@Service
public class TicketServiceImpl implements TicketService {

    @Override
    public TicketResponseDTO criar(TicketRequestDTO ticketRequestDTO)
        {
        Ticket ticket = mapper.mapTo(ticketRequestDTO, Ticket.class);
        Categoria categoria = categoriaService.buscaPorId(ticket.
           getCategoria().getId());
        Departamento departamento = mapper.mapTo(departamentoService.
           buscarPorId(ticket.getDepartamento().getId()),Departamento
           .class);
        RegraPrioridade regraPrioridade = regraPrioridadeService.
           buscarPorCategoriaAndDepartamento(categoria,departamento);

        Usuario cliente;

        if(ticket.getCliente() == null || ticket.getCliente().getId()
            == null){
            Long uuidAuth = Long.valueOf((String)
               SecurityContextHolder.getContext().getAuthentication()
               .getPrincipal());
            cliente = mapper.mapTo(usuarioService.buscaPorId(uuidAuth
               ),Usuario.class);
        } else
```

```
17            cliente = mapper.mapTo(usuarioService.buscaPorId(ticket.
                  getCliente().getId()),Usuario.class);
18
19        Usuario responsavel;
20
21        if(ticket.getResponsavel() == null || ticket.getResponsavel()
              .getId() == null)
22            responsavel = departamento.getGerente();
23        else
24            responsavel = ticket.getResponsavel();
25
26        ticket.setCliente(cliente);
27        ticket.setResponsavel(responsavel);
28        ticket.setDataCriacao(LocalDateTime.now());
29        ticket.setStatus(StatusTicket.ABERTO);
30        ticket.setCategoria(categoria);
31        ticket.setDepartamento(departamento);
32        ticket.setPrioridade(regraPrioridade.getPrioridade());
33        ticket.setDataMaximaResolucao(ticket.getDataCriacao().
              plusHours(regraPrioridade.getHorasResolucao()));
34        ticket = ticketRepository.save(ticket);
35        emailService.enviarTicketEmail(ticket);
36        return mapper.mapTo(ticket, TicketResponseDTO.class);
37    }
38
39    @Override
40    public List<TicketSimpleResponseDTO> buscarTodos() {
41        List<Ticket> ticketList = ticketRepository.findAll();
42
43        return mapper.toList(ticketList, TicketSimpleResponseDTO.
              class);
44    }
45
46    private Set<String> getUserRoles(Authentication authentication) {
47        return authentication.getAuthorities().stream()
48                .map(GrantedAuthority::getAuthority)
49                .collect(Collectors.toSet());
50    }
51
52    @Override
53    public List<TicketSimpleResponseDTO> buscarTodosFilter(String
          titulo, StatusTicket status, Prioridade prioridade, String
          nomeResponsavel, String dataInicio, String dataFim) {
54        Authentication authentication = SecurityContextHolder.
              getContext().getAuthentication();
55        Long uuidAuth = Long.valueOf((String) authentication.
              getPrincipal());
56        Set<String> roles = getUserRoles(authentication);
57
58        Specification<Ticket> spec = TicketSpecification.
              filterTickets(titulo, status, prioridade, nomeResponsavel,
```

```
            dataInicio,dataFim);
59
60          List<Ticket> ticketList = ticketRepository.findAll(spec);
61
62          if (roles.contains("ROLE_CLIENTE")) {
63              ticketList = ticketList.stream()
64                      .filter(ticket -> Objects.nonNull(ticket.
                           getCliente()) && Objects.equals(ticket.
                           getCliente().getId(), uuidAuth))
65                      .toList();
66          } else if (roles.contains("ROLE_FUNCIONARIO")) {
67              ticketList = ticketList.stream()
68                      .filter(ticket -> Objects.nonNull(ticket.
                           getResponsavel()) && Objects.equals(ticket.
                           getResponsavel().getId(), uuidAuth))
69                      .toList();
70          } else if (roles.contains("ROLE_GERENTE")) {
71              ticketList = ticketList.stream()
72                      .filter(ticket -> Objects.nonNull(ticket.
                           getDepartamento()) && Objects.equals(ticket.
                           getDepartamento().getGerente().getId(),
                           uuidAuth))
73                      .toList();
74          }
75
76          return mapper.toList(ticketList, TicketSimpleResponseDTO.
               class);
77      }
78
79      @Override
80      public TicketResponseDTO buscarPorId(Long id) {
81          Authentication authentication = SecurityContextHolder.
               getContext().getAuthentication();
82          Long uuidAuth = Long.valueOf((String) authentication.
               getPrincipal());
83          Set<String> roles = getUserRoles(authentication);
84
85          Optional<Ticket> ticketOptional;
86          if (roles.contains("ROLE_CLIENTE")) {
87              ticketOptional = ticketRepository.findByClienteIdAndId(
                   uuidAuth,id);
88          } else if (roles.contains("ROLE_FUNCIONARIO")) {
89              ticketOptional = ticketRepository.
                   findByResponsavelIdAndId(uuidAuth,id);
90          } else if (roles.contains("ROLE_GERENTE")) {
91              ticketOptional = ticketRepository.
                   findByDepartamentoGerenteIdAndId(uuidAuth,id);
92          } else
93              ticketOptional = ticketRepository.findById(id);
94
95          Ticket ticket = ticketOptional.orElseThrow(() ->
```

```
96                  new ResourceNotFoundException("Nao foi encontrado
                        nenhum ticket com esse id.")
97          );
98
99          return mapper.mapTo(ticket, TicketResponseDTO.class);
100     }
101
102     @Override
103     public TicketResponseDTO atualizar(Long id,
            TicketRequestUpdateDTO ticketRequestUpdateDTO) {
104         Authentication authentication = SecurityContextHolder.
                getContext().getAuthentication();
105         Long uuidAuth = Long.valueOf((String) authentication.
                getPrincipal());
106         Set<String> roles = getUserRoles(authentication);
107
108         Optional<Ticket> ticketOptional;
109         if (roles.contains("ROLE_CLIENTE")) {
110             ticketOptional = ticketRepository.findByClienteIdAndId(
                    uuidAuth,id);
111         } else if (roles.contains("ROLE_FUNCIONARIO")) {
112             ticketOptional = ticketRepository.
                    findByResponsavelIdAndId(uuidAuth,id);
113         } else if (roles.contains("ROLE_GERENTE")) {
114             ticketOptional = ticketRepository.
                    findByDepartamentoGerenteIdAndId(uuidAuth,id);
115         } else
116             ticketOptional = ticketRepository.findById(id);
117
118         Ticket ticket = ticketOptional.orElseThrow(() ->
119                 new ResourceNotFoundException("Nao foi encontrado
                        nenhum ticket com esse id.")
120         );
121
122         Map<String, Map<String, Object>> camposAlterados = new
                HashMap<>();
123
124         checkAndRecordEntityChange("categoria", ticket.getCategoria()
                , ticketRequestUpdateDTO.getCategoria(), camposAlterados);
125         checkAndRecordEntityChange("departamento", ticket.
                getDepartamento(), ticketRequestUpdateDTO.getDepartamento
                (), camposAlterados);
126         checkAndRecordEntityChange("responsavel", ticket.
                getResponsavel(), ticketRequestUpdateDTO.getResponsavel(),
                 camposAlterados);
127         List<String> variaveisIgnoradas = Arrays.asList("comentarios"
                , "id", "categoria", "departamento", "responsavel", "
                cliente", "class", "historicos", "registroTrabalhos");
128         BeanWrapper wrapper = new BeanWrapperImpl(ticket);
129         for (PropertyDescriptor descriptor : wrapper.
                getPropertyDescriptors()) {
```

```
130             String nomeVariavel = descriptor.getName();
131             if (!variaveisIgnoradas.contains(nomeVariavel)) {
132                 Object antigoValor = wrapper.getPropertyValue(
                        nomeVariavel);
133                 Object novoValor = new BeanWrapperImpl(
                        ticketRequestUpdateDTO).getPropertyValue(
                        nomeVariavel);
134
135                 if (novoValor != null && !novoValor.equals(
                        antigoValor)) {
136                     Map<String, Object> values = new HashMap<>();
137                     values.put("antigoValor", antigoValor);
138                     values.put("novoValor", novoValor);
139                     camposAlterados.put(nomeVariavel, values);
140                 }
141             }
142         }
143
144         Usuario usuarioResponsavelPelaAtualizacao = mapper.mapTo(
                usuarioService.buscaPorId(uuidAuth), Usuario.class);
145
146         List<TicketHistorico> ticketHistoricoList = new ArrayList<>()
                ;
147         camposAlterados.forEach((campo, valores) -> {
148                     Object antigoValor = valores.get("antigoValor");
149                     Object novoValor = valores.get("novoValor");
150                     TicketHistorico ticketHistorico = new
                            TicketHistorico();
151                     ticketHistorico.setCampo(campo);
152                     ticketHistorico.setUltimoValor(antigoValor.
                            toString());
153                     ticketHistorico.setNovoValor(novoValor.toString()
                            );
154                     ticketHistorico.setDataAlteracao(LocalDateTime.
                            now());
155                     ticketHistorico.setTicket(ticket);
156                     ticketHistorico.setAgente(
                            usuarioResponsavelPelaAtualizacao);
157                     ticketHistoricoList.add(ticketHistorico);
158                 }
159         );
160         ticket.getHistoricos().addAll(ticketHistoricoList);
161         ticketHistoricoList.forEach(ticketHistorico ->
                ticketHistoricoService.criar(ticketHistorico));
162
163         BeanUtils.copyProperties(ticketRequestUpdateDTO, ticket,
                objectUtils.getNullPropertyNames(ticketRequestUpdateDTO));
164
165         var ticketSalvado = ticketRepository.save(ticket);
166
167         if(ticketSalvado.getStatus() == StatusTicket.FINALIZADO)
```

```
168             emailService.enviarTicketFinalizadoEmail(ticketSalvado);
169
170         return mapper.mapTo(ticketSalvado, TicketResponseDTO.class);
171     }
172
173     @Override
174     public Ticket atualizaTicketEmAndamento(Usuario usuario, Ticket
            ticket){
175         String valorAntigo = ticket.getStatus().toString();
176         ticket.setStatus(StatusTicket.EM_ANDAMENTO);
177
178         List<TicketHistorico> ticketHistoricoList = new ArrayList<>()
                ;
179         TicketHistorico ticketHistorico = new TicketHistorico();
180         ticketHistorico.setCampo("status");
181         ticketHistorico.setUltimoValor(valorAntigo);
182         ticketHistorico.setNovoValor(ticket.getStatus().toString());
183         ticketHistorico.setDataAlteracao(LocalDateTime.now());
184         ticketHistorico.setTicket(ticket);
185         ticketHistorico.setAgente(usuario);
186         ticketHistoricoList.add(ticketHistorico);
187
188         ticket.getHistoricos().addAll(ticketHistoricoList);
189         ticketHistoricoList.forEach(ticketHistoricoCriar ->
                ticketHistoricoService.criar(ticketHistoricoCriar));
190
191         emailService.enviarTicketEmAndamentoEmail(ticket);
192
193         return ticketRepository.save(ticket);
194     }
195
196     private <T> void checkAndRecordEntityChange(String fieldName, T
            currentEntity, T newEntity, Map<String, Map<String, Object>>
            alteredFields) {
197         Long currentId = getEntityId(currentEntity);
198         Long newId = getEntityId(newEntity);
199
200         if (!Objects.equals(currentId, newId)) {
201             Map<String, Object> values = new HashMap<>();
202             values.put("antigoValor", currentId);
203             values.put("novoValor", newId);
204             alteredFields.put(fieldName, values);
205         }
206     }
207
208     private Long getEntityId(Object entity) {
209         if (entity == null) return null;
210         try {
211             Method getIdMethod = entity.getClass().getMethod("getId")
                    ;
212             return (Long) getIdMethod.invoke(entity);
```

```
213         } catch (NoSuchMethodException | IllegalAccessException |
                InvocationTargetException e) {
214             throw new RuntimeException("Erro ao obter o ID da
                    entidade", e);
215         }
216     }
217
218     @Override
219     public void deletePorId(Long id) {
220         Authentication authentication = SecurityContextHolder.
                getContext().getAuthentication();
221         Long uuidAuth = Long.valueOf((String) authentication.
                getPrincipal());
222         Set<String> roles = getUserRoles(authentication);
223
224         Optional<Ticket> ticketOptional;
225         if (roles.contains("ROLE_CLIENTE")) {
226             ticketOptional = ticketRepository.findByClienteIdAndId(
                    uuidAuth,id);
227         } else if (roles.contains("ROLE_FUNCIONARIO")) {
228             ticketOptional = ticketRepository.
                    findByResponsavelIdAndId(uuidAuth,id);
229         } else if (roles.contains("ROLE_GERENTE")) {
230             ticketOptional = ticketRepository.
                    findByDepartamentoGerenteIdAndId(uuidAuth,id);
231         } else
232             ticketOptional = ticketRepository.findById(id);
233
234         if(ticketOptional.isPresent())
235             ticketRepository.deleteById(id);
236     }
237 }
```

Listing 4.4: Ticket

4.5.5 Registro de Trabalho

O Listing 4.5 apresenta a implementação do serviço responsável pelo gerenciamento dos registros de trabalho no sistema. A classe RegistroTrabalhoServiceImpl contém métodos para criação, busca, atualização e remoção desses registros, garantindo o vínculo com *tickets* e usuários do sistema.

O método criar recebe um RegistroTrabalhoRequestDTO e associa o registro a um *ticket* específico e ao usuário autenticado. Caso o *ticket* ainda esteja no status "Aberto", ele é atualizado para "Em Andamento"após a criação do registro.

O serviço também permite a busca de todos os registros de trabalho relacionados a um *ticket* por meio do método buscarTodosPorTicket, retornando uma lista de registros vinculados ao chamado.

A funcionalidade de atualização (atualizar) permite modificar um registro existente, garantindo que apenas os campos não nulos sejam alterados, enquanto o método deletarPorId possibilita a remoção de registros de trabalho pelo seu identificador.

Essa implementação garante a rastreabilidade das atividades realizadas em cada *ticket*, auxiliando no controle e na gestão dos chamados dentro do sistema.

```
1  public class RegistroTrabalhoServiceImpl implements
       RegistroTrabalhoService {
2      @Override
3      public RegistroTrabalhoReponseDTO criar(Long ticketId,
           RegistroTrabalhoRequestDTO registroTrabalhoRequestDTO) {
4          Ticket ticket = mapper.mapTo(ticketService.buscarPorId(
               ticketId), Ticket.class);
5          Long uuidAuth = Long.valueOf((String) SecurityContextHolder.
               getContext().getAuthentication().getPrincipal());
6          Usuario usuario = mapper.mapTo(usuarioService.buscaPorId(
               uuidAuth), Usuario.class);
7
8          RegistroTrabalho registroTrabalho = mapper.mapTo(
               registroTrabalhoRequestDTO,RegistroTrabalho.class);
9          registroTrabalho.setTicket(ticket);
10         registroTrabalho.setAgente(usuario);
11
12         var registro = mapper.mapTo(registroTrabalhoRepository.save(
               registroTrabalho), RegistroTrabalhoReponseDTO.class);
13
14         if(ticket.getStatus() == StatusTicket.ABERTO && registro.
               getId() != null)
15             ticket = ticketService.atualizaTicketEmAndamento(usuario,
                   ticket);
16
```

```
17          registro.setTicket(mapper.mapTo(ticket,
                TicketSimpleResponseDTO.class));
18
19          return registro;
20      }
21
22      @Override
23      public List<RegistroTrabalhoReponseDTO> buscarTodosPorTicket(Long
            ticketId) {
24          return mapper.toList(registroTrabalhoRepository.
                findByTicketId(ticketId), RegistroTrabalhoReponseDTO.class
                );
25      }
26
27      @Override
28      public RegistroTrabalhoReponseDTO atualizar(Long registroId,
            RegistroTrabalhoRequestUpdateDTO
            registroTrabalhoRequestUpdateDTO) {
29          RegistroTrabalho registroTrabalho =
                registroTrabalhoRepository.findById(registroId)
30                  .orElseThrow(() -> new ResourceNotFoundException("Nao
                        foi encontrado nenhum registro com esse id."));
31          BeanUtils.copyProperties(registroTrabalhoRequestUpdateDTO,
                registroTrabalho,objectUtils.getNullPropertyNames(
                registroTrabalhoRequestUpdateDTO));
32          return mapper.mapTo(registroTrabalhoRepository.save(
                registroTrabalho), RegistroTrabalhoReponseDTO.class);
33      }
34
35      @Override
36      public void deletarPorId(Long registroId) {
37          registroTrabalhoRepository.deleteById(registroId);
38      }
39  }
```

Listing 4.5: Registro de Trabalho

4.6 Documentação Swagger do sistema

A documentação da API é um componente essencial para garantir a compreensibilidade, reutilização e manutenção do sistema. No desenvolvimento do sistema proposto, foi utilizada a biblioteca Swagger (atualmente mantida como OpenAPI) para geração automática da documentação da API REST.

O Swagger permite descrever, de forma padronizada, todos os endpoints disponíveis, bem como os métodos HTTP suportados, parâmetros esperados, códigos de resposta e exemplos de requisições e respostas. A documentação gerada é interativa e pode ser acessada via navegador,

permitindo testes diretos nos endpoints, o que facilita o trabalho de desenvolvedores e testadores.

A seguir, a Figura 4.3 e a Figura 4.4 apresenta um exemplo da interface Swagger gerada automaticamente:

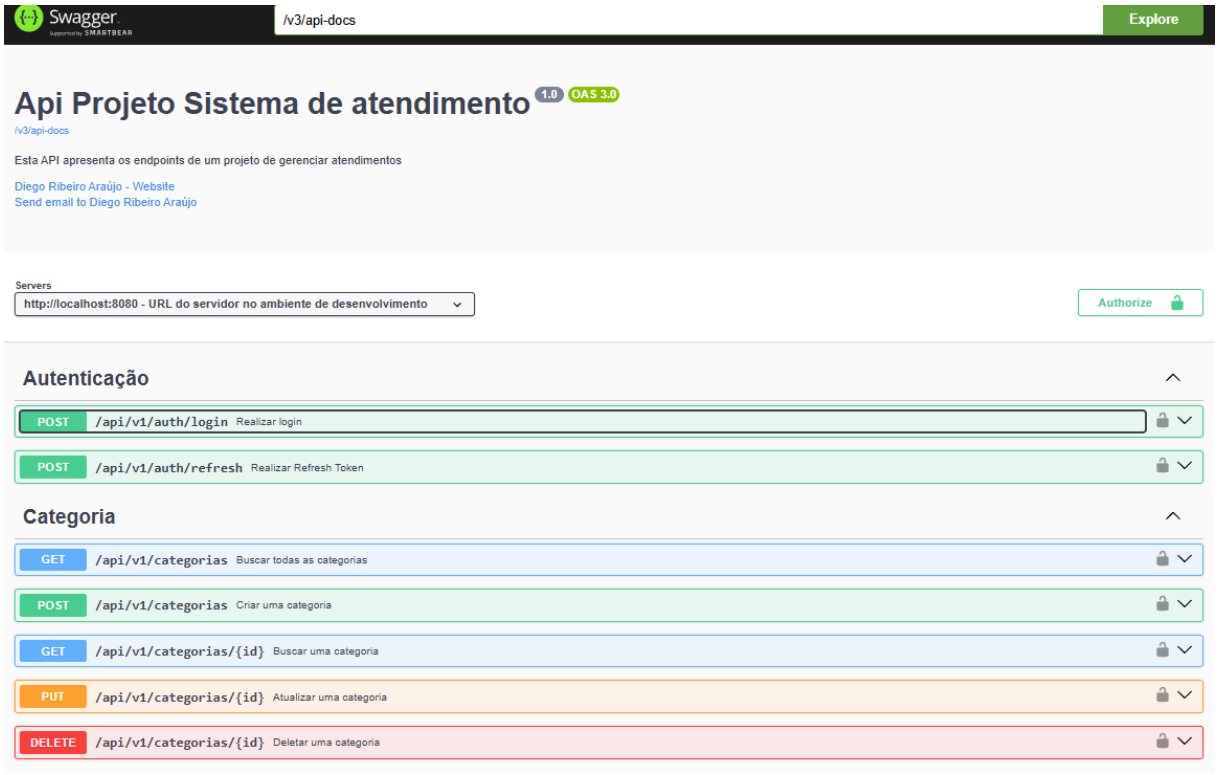


Figura 4.3: Visualização geral da interface do Swagger

Figura 4.4: Endpoints disponíveis na Api

Já a Figura 4.5 apresenta uma requisição de login realizada diretamente pela interface do Swagger, exibindo o corpo (*body*) enviado com as credenciais do usuário e a Figura 4.6 apresenta a resposta retornada pela API. Nesse caso, ao informar corretamente o e-mail e a senha, a aplicação retorna um token JWT, que poderá ser utilizado para autenticar as próximas requisições em rotas protegidas.

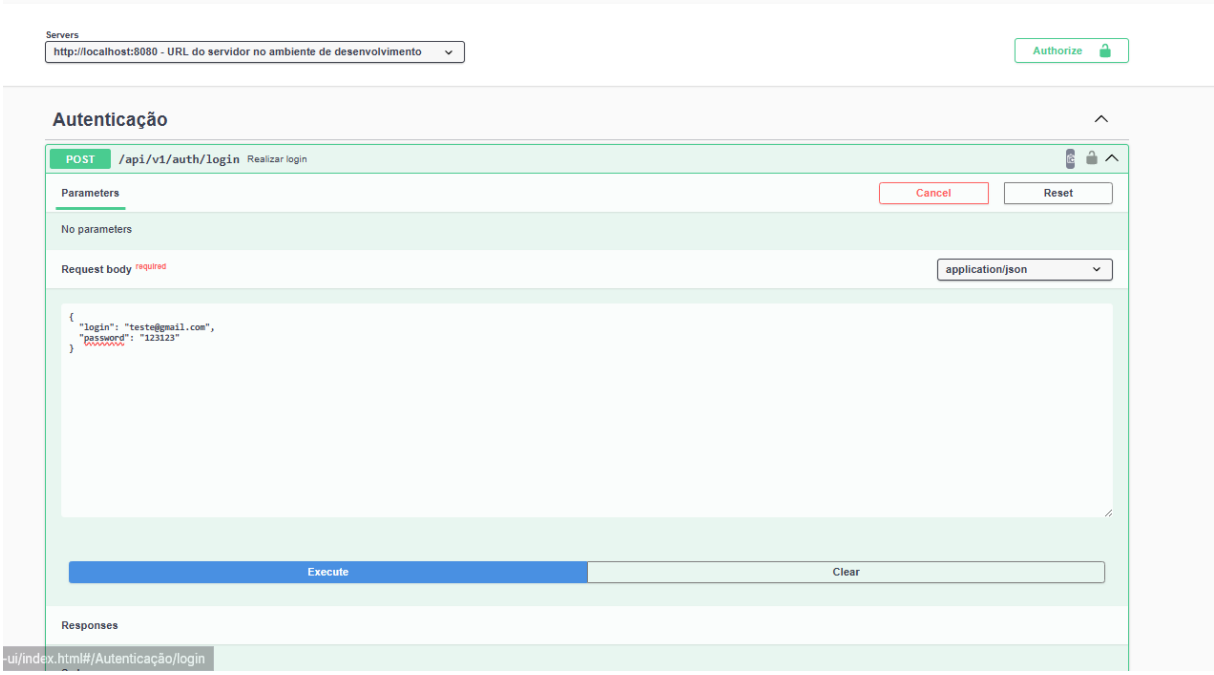


Figura 4.5: Requisição de login

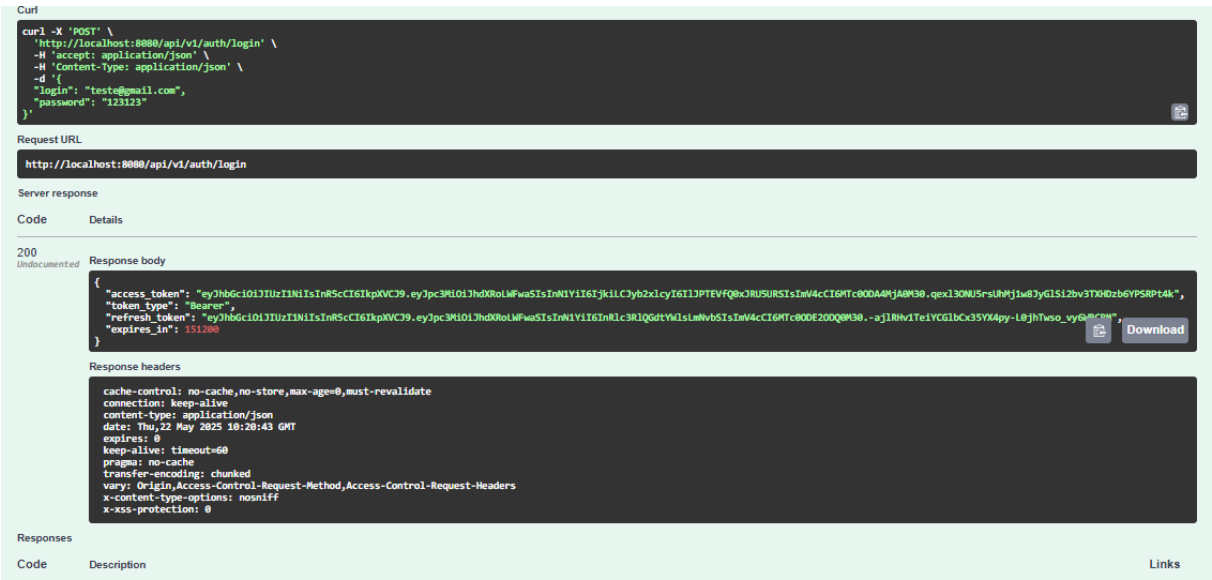


Figura 4.6: Resposta da requisição de login

CAPÍTULO 5

A APLICAÇÃO

Até o momento, foram apresentados diversos aspectos relacionados à implementação da API. Neste capítulo, será exposto o resultado final do sistema. A seguir, são demonstradas as chamadas da API, evidenciando funcionalidades como cadastro de usuários, autenticação (login) e criação de tickets. Essa implementação assegura a rastreabilidade das atividades realizadas em cada *ticket*, contribuindo significativamente para o controle e a gestão eficiente dos chamados no sistema.

5.1 Criar Usuário

A Figura 5.1 apresenta a tela do postman referente a criação de usuário no sistema. Esse recurso permite que novos usuários sejam cadastrados com seus dados pessoais e credenciais de acesso. No teste realizado, a funcionalidade de criação de usuário foi executada com sucesso, validando que o sistema está apto a registrar novos perfis e armazená-los corretamente no banco de dados.

Durante o teste, foram inseridas informações válidas em todos os campos obrigatórios do formulário, e após a submissão, o sistema retornou uma confirmação positiva, indicando que o usuário foi criado com êxito.

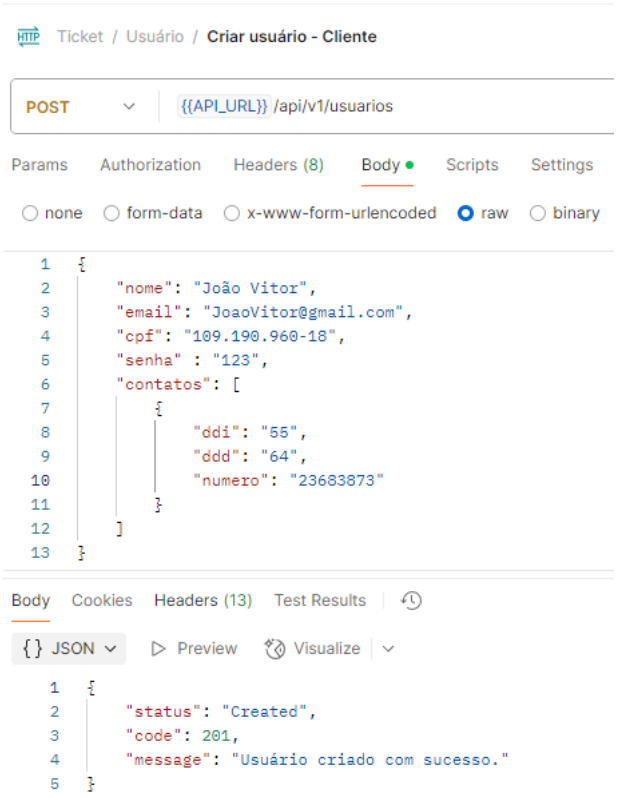
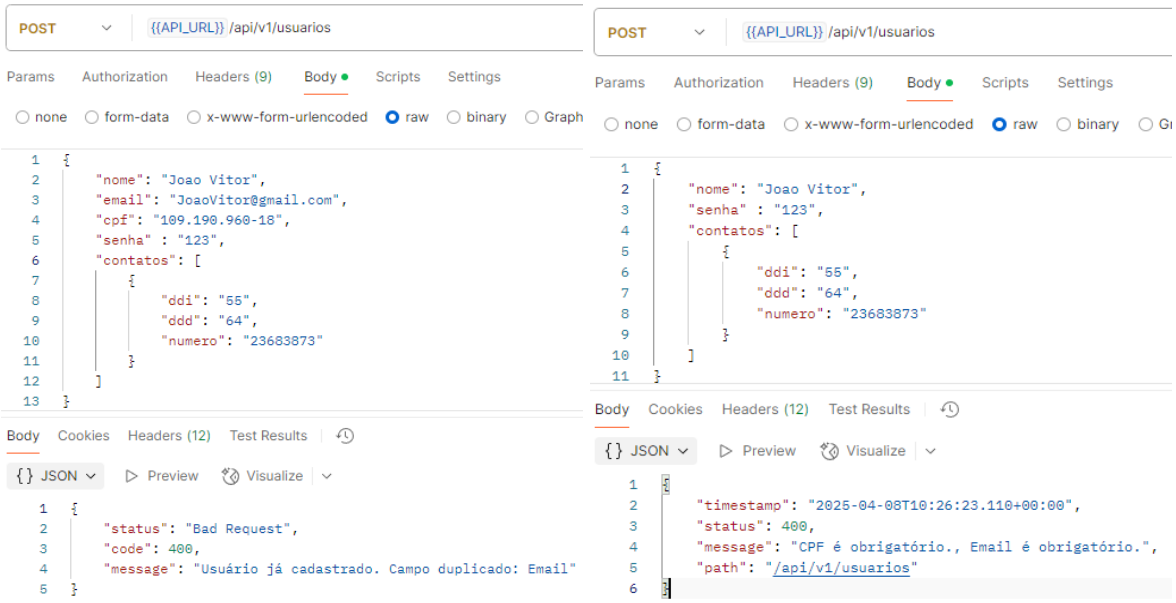


Figura 5.1: Criar Usuário

A Figura 5.2 apresenta dois tipos de erros que podem ocorrer durante o processo de criação de usuários no sistema. Na subfigura 5.2a, é exibida a mensagem de erro gerada quando o email de usuário informado já está cadastrado no sistema, impedindo a duplicidade de registros. Já a subfigura 5.2b mostra a mensagem apresentada quando o formulário é submetido com campos obrigatórios não preenchidos, o que reforça a necessidade de validação dos dados inseridos pelo usuário antes do envio.



(a) Erro de usuário duplicado **(b)** Erro de campos ausentes

Figura 5.2: Erros na criação de usuários

5.2 Realizar Login

A Figura 5.3 apresenta a tela do postman para realizar o login do usuário no sistema. Ao informar o e-mail e a senha previamente cadastrados, o sistema valida as credenciais, retornando um token de acesso (JWT) com validade de 1 hora, além de um refresh token com validade de 8 horas. Esses mecanismos garantem a segurança do acesso às funcionalidades do sistema, permitindo a renovação da sessão de forma transparente e controlada. O teste confirmou que o fluxo de autenticação está funcionando corretamente, proporcionando uma experiência segura e eficiente para o usuário.

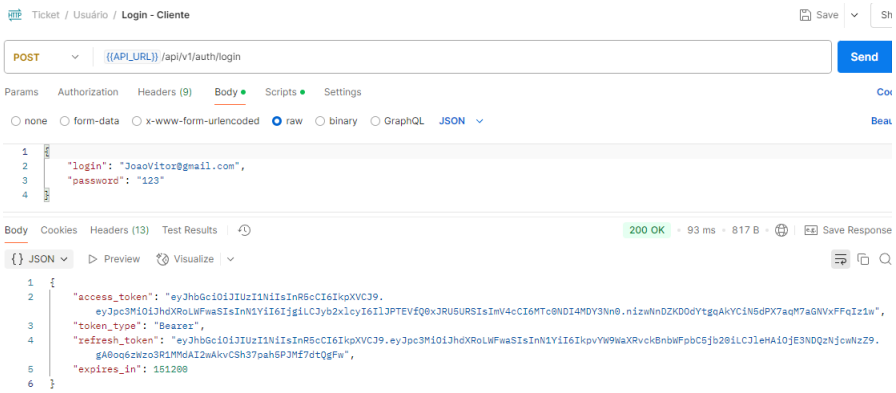


Figura 5.3: Realizar Login

5.3 Criar *Ticket*

A Figura 5.4 apresenta a interface do postman para a funcionalidade de criação de *tickets* no sistema. Essa funcionalidade é fundamental para registrar e gerenciar solicitações feitas pelos usuários. Durante o teste, foi preenchido o formulário com os dados obrigatórios para a criação do *ticket*, como o título, que resume a solicitação, e a descrição, onde são detalhadas as informações do problema ou da demanda a ser atendida, além de informar também o departamento e categoria do chamado.

Após o preenchimento correto dos campos, o usuário confirmou a criação do *ticket*, e o sistema respondeu com uma mensagem de sucesso, indicando que o registro foi armazenado com êxito no banco de dados. Esse teste validou que a API está processando corretamente os dados enviados, vinculando o *ticket* ao usuário autenticado e disponibilizando-o para acompanhamento e resolução futura.

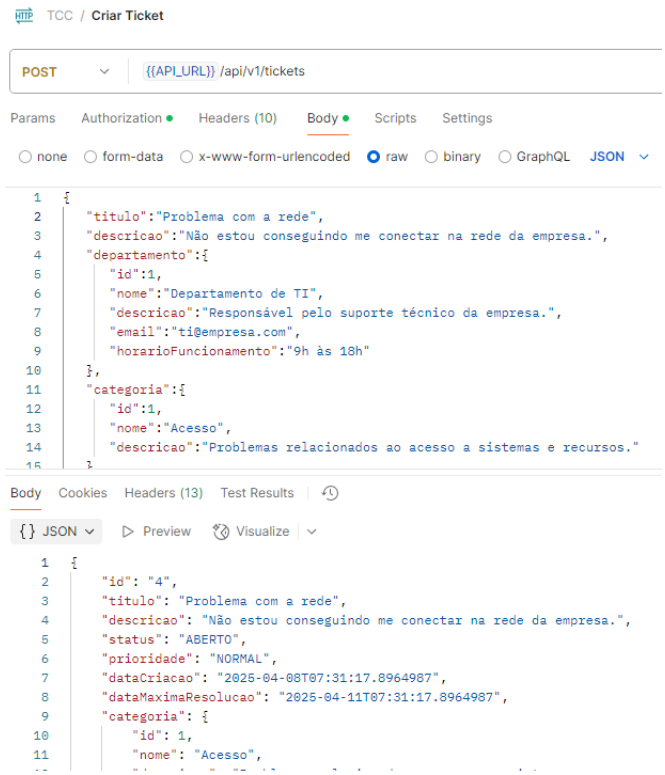


Figura 5.4: Criar *Ticket*

5.4 Registrar Atendimento

A Figura 5.5 apresenta a interface de registro de atendimento de um *ticket*. Essa funcionalidade é essencial para documentar as ações realizadas durante o processo de suporte, permitindo

maior controle sobre o histórico de cada solicitação. Ao iniciar um atendimento, o sistema registra automaticamente o horário de início, e, ao finalizá-lo, calcula o tempo total gasto com base na diferença entre os horários de início e término.

Durante o teste, um atendimento foi iniciado a partir de um *ticket* previamente criado. Após a inserção de uma descrição com os detalhes das ações executadas, o atendimento foi finalizado com sucesso. O sistema então apresentou o tempo decorrido, confirmando que o cálculo de duração do atendimento está funcionando corretamente. Esse recurso é importante tanto para a avaliação da eficiência da equipe quanto para a geração de indicadores de desempenho e relatórios gerenciais.

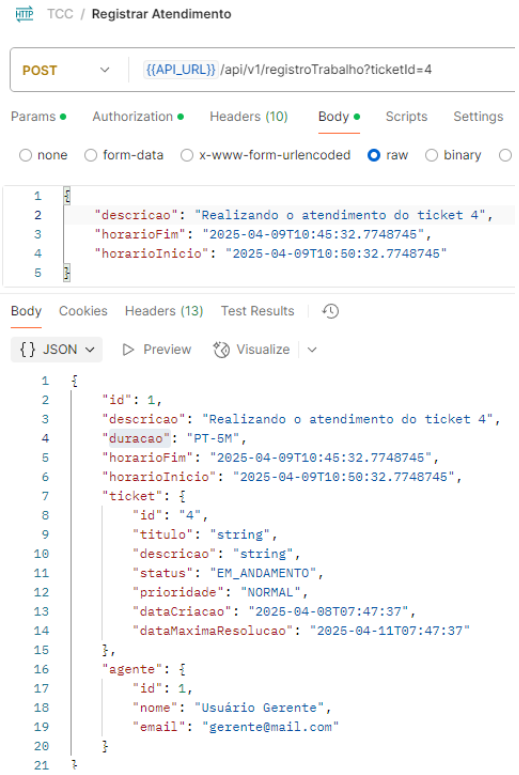


Figura 5.5: Registrar atendimento de algum *ticket*

5.5 Criar Comentário

A Figura 5.6 exibe a funcionalidade de criação de comentários em um *ticket*. Esse recurso tem como objetivo registrar observações, atualizações ou interações complementares ao atendimento principal, facilitando a comunicação entre os usuários envolvidos e mantendo um histórico detalhado do progresso da solicitação.

Durante o teste, foi inserido um comentário em um *ticket* já existente. O sistema exigiu que o campo de texto fosse preenchido com uma mensagem válida, e após a submissão,

confirmou o registro do comentário com sucesso. A funcionalidade demonstrou estar funcionando corretamente, associando o conteúdo ao *ticket*.

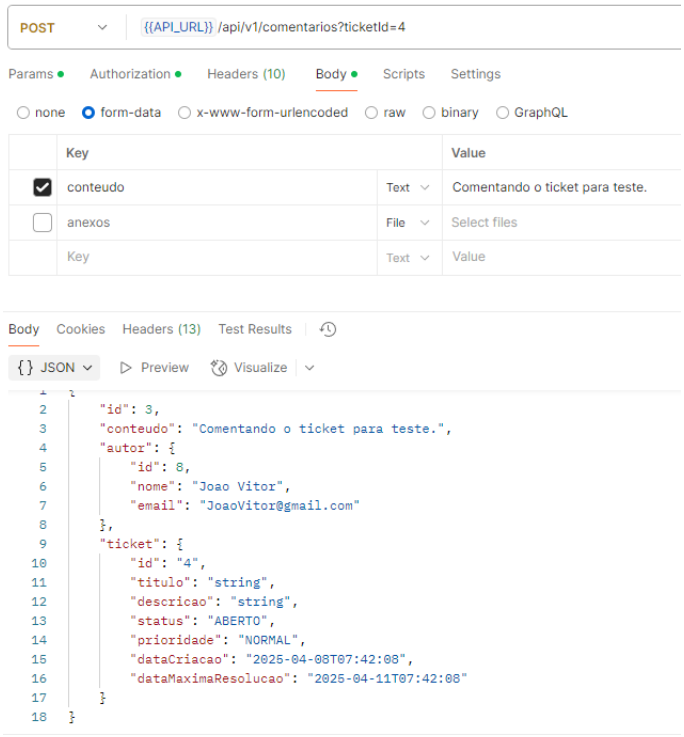


Figura 5.6: Criar Comentário de um *ticket*

CONCLUSÃO

Este trabalho teve como objetivo o desenvolvimento de uma API para gerenciamento de *tickets* de atendimento, visando atender às necessidades de empresas que buscam melhorar seus processos internos de suporte e relacionamento com clientes. A solução proposta se destacou por sua flexibilidade, segurança e capacidade de personalização, oferecendo um sistema capaz de se adaptar às diferentes realidades organizacionais.

Ao longo do desenvolvimento, foram aplicados conceitos sólidos de engenharia de software, como a utilização de autenticação com JSON Web Token (JWT), arquitetura modular e boas práticas de segurança e escalabilidade. Além disso, o sistema foi estruturado para permitir integrações com outros serviços corporativos, oferecendo recursos como categorização de chamados, definição de níveis de prioridade, notificações automáticas e controle de fluxo de atendimento.

Os testes realizados demonstraram a eficácia da aplicação em cenários reais de uso, com funcionalidades completas de criação, atualização, listagem e exclusão de *tickets*, bem como o registro de atividades relacionadas. A modelagem da base de dados, aliada ao uso de frameworks robustos, garantiu um desempenho satisfatório mesmo com grandes volumes de dados.

A Tabela 5.1 apresenta um comparativo entre o sistema desenvolvido e três das principais ferramentas comerciais de gerenciamento de *tickets* (Zendesk, Freshdesk e Zoho Desk), destacando-se aspectos como capacidade de personalização, integração com sistemas legados, controle de acesso e custo. Os dados demonstram que a solução criada oferece vantagens competitivas relevantes, especialmente para organizações que demandam maior flexibilidade e adaptação a contextos específicos.

Característica	Zendesk	Freshdesk	Zoho Desk	Sistema criado
Interface amigável e intuitiva	Sim	Sim	Sim	Sim
Customização de fluxos e regras	Alta	Baixa	Média	Alta
Integração com sistemas legados	Limitada	Limitada	Limitada	Avançada
Automação de *tickets* e respostas	Sim	Sim	Sim	Sim
Gerenciamento de SLA e métricas	Sim	Sim	Sim	Sim
Controle de acesso e permissões	Limitado	Limitado	Limitado	Avançado
Preço	Caro	Mais acessível	Mais acessível	Mais acessível

Tabela 5.1: Comparativo de Características em Sistemas de Gerenciamento de *Tickets* para o sistema criado

Como trabalhos futuros, poderá ser realizada a implementação de novas funcionalidades no sistema, como um dashboard analítico com gráficos e indicadores que permitam a visualização em tempo real dos *tickets* abertos, em andamento, resolvidos, tempo médio de atendimento, entre outros dados relevantes para a gestão de desempenho das equipes de suporte. Também é proposta a integração com um chatbot e a criação de um módulo de feedback, permitindo que o cliente avalie o atendimento após o encerramento do *ticket*, contribuindo para a melhoria contínua do serviço.

No entanto, este trabalho apresenta algumas limitações. Por se tratar de uma aplicação do tipo API, não foi possível realizar testes em um ambiente real de produção, uma vez que seria necessário disponibilizar a aplicação em um servidor para simular o uso contínuo. Além disso, a ausência de uma interface gráfica (front-end) dificultou a validação do sistema por usuários finais, restringindo os testes à utilização de ferramentas como o Postman ou o Swagger UI, o que limita a avaliação da usabilidade e da experiência do usuário. A criação de uma aplicação front-end integrada é, portanto, um passo essencial para uma avaliação mais completa da solução proposta.

REFERÊNCIAS BIBLIOGRÁFICAS

AMUNDSEN, M. *RESTful Web API Patterns and Practices Cookbook*. [S.l.]: O'Reilly Media, 2022. ISBN 1098106741.

AXELOS. *ITIL Foundation: ITIL 4 Edition*. [S.l.]: TSO (The Stationery Office), 2019. ISBN 0113316070.

CHAPMAN, N. *API Security in Action*. [S.l.]: Manning Publications, 2020. ISBN 1617296023.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000.

FITZSIMMONS, J.; FITZSIMMONS, M. *Service Management: Operations, Strategy, and Information Technology*. [S.l.]: McGraw-Hill, 2023. ISBN 978-9355324870.

Freshdesk. *Freshdesk - Helpdesk Software Features*. 2010. Accessed: 14 Jul. 2025. Disponível em: <https://www.freshworks.com/br/freshdesk/>.

GEEWAX, J. J. *API Design Patterns*. [S.l.]: Manning Publications, 2021. ISBN 9781617295850.

STALLINGS, W. *Cryptography and Network Security Principles and Practice*. [S.l.]: Pearson Education, 2023. ISBN 1-292-43748-0.

TALEBI, H.; BARDSIRI, A. K. The impact of information technology on service quality, satisfaction, and customer relationship management (case study: It organization individuals). *Journal of Management Science & Engineering Research*, v. 6, n. 2, p. 24–31, Sep. 2023. Disponível em: <https://journals.bilpubgroup.com/index.php/jmser/article/view/5823>.

Zendesk. *Zendesk Features Overview*. 2007. Accessed: 14 Jul. 2025. Disponível em: <https://www.zendesk.com.br>.

Zoho Desk. *Zoho Desk - Customer Service Software*. 2010. Accessed: 14 Jul. 2025. Disponível em: <https://www.zoho.com/pt-br/desk/>.