UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Gustavo Vicenzi

Desenvolvimento de Sistema de Gerenciamento de Reservas e Controle Automatizado de Dispositivos IoT para Quadras Esportivas

Blumenau
2025

Gustavo Vicenzi

Desenvolvimento de Sistema de Gerenciamento de Reservas e Controle Automatizado de Dispositivos IoT para Quadras Esportivas

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.
Orientador: Prof. Carlos Barros Montez, Dr.
Supervisor: Matheus Fischer, Eng.

Blumenau
2025

Ficha de identificação da obra

A ficha de identificação é elaborada pelo próprio autor.

Orientações em:

http://portalbu.ufsc.br/ficha

Gustavo Vicenzi

Desenvolvimento de Sistema de Gerenciamento de Reservas e Controle Automatizado de Dispositivos IoT para Quadras Esportivas

Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação

Florianópolis, <dia(número)> de <mês> de <ano(número)>.

Prof. Marcelo De Lellis Costa De Oliveira, Dr.
Coordenador do Curso

Banca Examinadora:
[preencher somente após a defesa para a Versão Final da BU]

Prof(a). Carlos Barros Montez, Dr.
Orientador(a)
UFSC/CTC/EAS

Matheus Fischer, Eng.
Supervisor(a)
Fischertec Tecnologia

Prof(a). xxxx, Dr(a).
Avaliador(a)
Instituição xxxx

Prof. xxxx, Dr.
Presidente da Banca
UFSC/CTC/EAS

Este trabalho é dedicado aos meus amigos, colegas de classe, namorada e aos meus queridos pais.

AGRADECIMENTOS

Primeiramente, expresso minha gratidão à minha família, que sempre me incentivou a seguir meus sonhos acadêmicos. Agradeço particularmente à minha mãe Sandra Vicenzi e meu pai Sergio Vicenzi, pelo seu amor incondicional e pela confiança depositada em mim. Também quero mencionar minha irmã Micheli e meu cunhado Matheus, que sempre me fornecem suporte e ajuda nas horas difíceis.

Agradeço também à minha avó Vilma e à minha tia Samara e tio Celso, que sempre me inspiraram a manter o foco e buscar excelência em tudo que faço. Por fim, quero destacar minha namorada Monike, que compartilha comigo todo o amor e paciência durante os momentos mais ansiosos da minha trajetória acadêmica. Seu apoio foi essencial para manter me motivado e garantir que todo o trabalho fosse feito com qualidade e entusiasmo.

Agradeço ao meus colegas de classe da turma da Automação 16.2, cuja amizade e colaboração fizeram toda a diferença no desenvolvimento deste projeto. Seus comentários e sugestões foram essenciais para refinar ideias e garantir que o trabalho fosse realizado com excelência. O apoio da turma foi fonte de inspiração para manter me motivado durante os momentos mais desafiadores.

Em seguida, quero expressar minha gratidão pelo incrível suporte recebido por meu orientador acadêmico, Carlos Montez. Sua sabedoria, paciência e expertise foram fundamentais para guiar meus passos acadêmicos, permitindo que eu avançasse com confiança no caminho da excelência na pesquisa e desenvolvimento.

Também quero agradecer ao meu supervisor Matheus Fischer por proporcionar o contexto prático da problemática estudada neste trabalho. Seu apoio foi crucial para garantir que a solução proposta estivesse alinhada com as necessidades reais da empresa e que fossem disponibilizadas todas as recursos necessários para a realização do projeto.

Por fim, agradeço a todos os meus professores da Universidade Federal de Santa Catarina, que sempre compartilharam seus conhecimentos, garantindo a qualidade e rigor dos meus estudos. Suas contribuições foram fundamentais para o sucesso deste projeto.

DECLARAÇÃO DE PUBLICIDADE

Blumenau, 17 de janeiro de 2025.

Na condição de representante da <instituição de realização do PFC> na qual o presente trabalho foi realizado, declaro não haver ressalvas quanto ao aspecto de sigilo ou propriedade intelectual sobre as informações contidas neste documento, que impeçam a sua publicação por parte da Universidade Federal de Santa Catarina (UFSC) para acesso pelo público em geral, incluindo a sua disponibilização *online* no Repositório Institucional da Biblioteca Universitária da UFSC. Além disso, declaro ciência de que o autor, na condição de estudante da UFSC, é obrigado a depositar este documento, por se tratar de um Trabalho de Conclusão de Curso, no referido Repositório Institucional, em atendimento à Resolução Normativa n° 126/2019/CUn.

Por estar de acordo com esses termos, subscrevo-me abaixo.

_____

Matheus Fischer, Eng.
Fischertec Tecnologia

RESUMO

A Fischertec Tecnologia é uma empresa brasileira que se concentra no desenvolvimento de soluções digitais de alta qualidade. Atua no setor de sites, aplicativos, e-commerces e diversos produtos digitais, buscando transformar ideias em soluções bem projetadas e inovadoras.

O principal objetivo deste PFC é criar um *Minimum Viable Product* (MVP) de uma plataforma de agendamento de reservas de quadras esportivas em uma aplicação *Software as a Service* (SaaS). A criação desse sistema foi motivada pelas demandas específicas das empresas esportivas, como personalização de horários, preços e integrações com dispositivos *Internet of Things* (IoT).

A Fischertec busca fornecer uma solução robusta, eficiente e alinhada às melhores práticas de automação e gestão esportiva. Ela optou por desenvolver um sistema proprietário que ofereça a experiência dos clientes empresariais com um produto único e personalizável, em busca da redução de sua dependência de projetos externos.

O desenvolvimento da aplicação do servidor, que é o escopo deste PFC, utiliza Nest.js para a API REST, PostgreSQL como banco de dados e Docker para conteinerização. O sistema frontend, em Next.js/React, será responsável pela interface do usuário em um futuro desenvolvimento. A metodologia utilizada foi ágil, garantindo entregas iterativas e validações frequentes com o TypeORM para persistência de dados. Postman serve como ferramenta de testes da API.

O principal resultado obtido é a entrega de uma aplicação backend totalmente funcional que permite o gerenciamento completo de locações, com autenticação e controle de usuários e papéis para diferentes *tenants* e preparada para uma futura integração com interface gráfica web. O sistema se mostrou eficiente, alinhado aos objetivos propostos e de fácil usabilidade.

A aplicação do servidor é totalmente original e inovadora, destacando-se na resolução dos problemas enfrentados pela Fischertec ao simular um cenário real de operação.

Palavras-chave: SaaS. Nest.js. IoT. Desenvolvimento. Software. Backend. Quadras. Esportivas. Gerenciamento. Agendamento

ABSTRACT

The Fischertec Tecnologia is a Brazilian company focused on providing high-quality digital solutions. It operates in the fields of websites, applications, e-commerce, and various digital products, with a focus on transforming ideas into well-designed and innovative solutions.

The main objective of this PFC is to create a *Minimum Viable Product* (MVP) of a sports fields reservations management platform in a *Software as a Service* (SaaS) environment. The creation of this system was motivated by the specific needs of sports industry companies, such as customizable schedules, prices, and IoT device integrations.

The Fischertec company aims to provide a robust, efficient, and aligned with best practices solution for sports enthusiasts by offering an unique and personalized product experience. They opted to develop an owner-owned solution that aims to reduce their dependency on external projects. During this PFC project, the server application development scope is focused. It uses Nest.js for the REST API, PostgreSQL as the database, and Docker for containerization. The frontend system in Next.js/React will be responsible for the graphical user interface in a future development. The methodology used was agile, ensuring iterative deliveries and frequent validations with TypeORM for data persistence. Postman is used as a tool for testing the API. The main result obtained is the delivery of a fully functional backend application that allows comprehensive management and booking/scheduling of sports locations, with authentication and user roles for different tenants. The system was efficient and aligned with the proposed objectives and easy to use.

The server-side application is original and innovative, standing out in the resolution of issues faced by the Fischertec company through simulating a real operation scenario.

Keywords: SaaS. Nest.js. IoT. Development. Software. Backend. Fields. Sports. Management. Booking. Scheduling.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
IoT	*Internet of Things*
MVP	*Minimum Viable Product*
OAuth	*Open Authorization*
REST	*Representational State Transfer*
SaaS	*Software as a Service*

SUMÁRIO

1 INTRODUÇÃO

A situação atual das empresas que administram quadras esportivas revela uma crescente demanda por sistemas de agendamento e controle de reservas mais eficientes e automatizados. Com o aumento da competitividade no setor, essas empresas buscam soluções que não apenas organizem os horários e disponibilidades de seus espaços, mas que também integrem funcionalidades de automação de dispositivos, como iluminação, climatização e sistemas de acesso. A automação integrada aos agendamentos é crucial para reduzir desperdícios de energia e otimizar os custos operacionais, permitindo que os dispositivos sejam acionados apenas nos horários em que as quadras estão efetivamente em uso. Além disso, essas práticas têm um impacto positivo no meio ambiente, reduzindo a pegada de carbono das operações e promovendo um modelo de negócio mais sustentável.

1.1 A EMPRESA FISCHERTEC

A Fischertec Tecnologia é uma empresa dedicada a transformar ideias em soluções digitais de alta qualidade. Atua no desenvolvimento de sites, aplicativos, ecommerces e diversos produtos digitais, sempre focando em inovação e excelência.

A empresa conta com uma equipe de especialistas em design e desenvolvimento. Os designers são experientes em UX/UI e dedicados a criar interfaces que proporcionem a melhor experiência ao usuário. Os desenvolvedores fullstack da equipe da Fischertec têm expertise em diversas tecnologias e frameworks, garantindo a entrega de soluções robustas e eficazes.

A Fischertec, buscando criar um produto próprio com o objetivo de reduzir sua dependência de projetos externos, optou por desenvolver um sistema de gerenciamento de reservas de quadras esportivas. A decisão de desenvolver esse sistema foi motivada pela necessidade de atender demandas específicas das empresas esportivas, como personalização de horários, preços e integrações com dispositivos IoT. A criação de um sistema proprietário também reflete a estratégia da Fischertec de se posicionar como referência no setor, oferecendo uma solução robusta, eficiente e alinhada às melhores práticas de automação e gestão esportiva.

1.2 O PROBLEMA E OS OBJETIVOS

O problema central abordado é a dificuldade em gerenciar reservas de forma integrada e eficiente, definindo horários de funcionamento, preços de locação, tipos de quadras disponíveis entre outras opções. Essa dificuldade impacta diretamente a experiência dos clientes, bem como a eficiência administrativa da empresa.

O objetivo geral do projeto foi desenvolver um *Minimum Viable Product* (MVP)

de uma plataforma que permita gerenciar reservas de quadras esportivas de forma centralizada, disponível na web e personalizável para diferentes empresas. Foram estabelecidos alguns objetivos específicos:

- A criação de um sistema com autenticação de usuários;
- Controle de permissões de usuário baseado em papéis e configurável para diferentes *tenants*;
- Definição e gerenciamento de quadras com possibilidade de personalização, como tipos de quadras, preços e horários de funcionamento;
- Configuração do *tenant* da empresa, com nome e horários de funcionamento;
- Agendamento de reservas com controle de disponibilidade;
- Integração com dispositivos IoT para automação de recursos presentes nas quadras de acordo com as reservas.

A Fischertec busca fornecer uma solução que otimize os processos operacionais e ofereça uma experiência mais eficiente e moderna para seus clientes empresariais, com foco nos esportistas que utilizam quadras esportivas e participam de aulas coletivas. Portanto, foi proposta uma solução que envolveu o desenvolvimento de uma aplicação web composta por um servidor com API REST e um sistema frontend, ambos projetados para oferecer flexibilidade e desempenho. O sistema foi implementado utilizando uma arquitetura modular baseada em camadas, promovendo a separação de responsabilidades e facilitando futuras manutenções e expansões. Foram utilizadas tecnologias modernas, como Nest.js para a API, PostgreSQL para o banco de dados e Docker para conteinerização. A interface web será futuramente desenvolvida em Next.js/React. Além disso, houve integração com a plataforma IoT eWeLink da Shenzhen Coolkit Technology CO., LTD, possibilitando o controle automatizado de dispositivos associados às quadras.

A metodologia aplicada ao desenvolvimento seguiu práticas ágeis, garantindo entregas iterativas e validações frequentes. Ferramentas como Postman foram utilizadas para testes das APIs, enquanto o TypeORM auxiliou na persistência dos dados. O desenvolvimento foi estruturado de modo a criar um sistema totalmente funcional e alinhado com os objetivos propostos. Devido à maior complexidade, optou-se por focar no desenvolvimento da aplicação do servidor, que é o escopo deste PFC.

Os principais resultados obtidos incluem a entrega de uma aplicação backend que permite o gerenciamento completo de locações, com controle de usuários e papéis para diferentes *tentans*, gerenciamento de horários e automação de dispositivos IoT no início e fim das reservas. O sistema se mostrou eficiente e de fácil usabilidade, atendendo às expectativas da Fischertec ao simular um cenário real de operação.

Todo o desenvolvimento da aplicação do servidor foi realizado pelo autor, sem aproveitamento direto de trabalhos prévios ou de outros times, o que reforça o caráter original e inovador do projeto. A aplicação cumpre o objetivo de resolver os problemas de gerenciamento enfrentados pela Fischertec, destacando-se como uma solução escalável e tecnológica para o setor.

1.3 ESTRUTURA DO DOCUMENTO

O presente documento está organizado da seguinte maneira. O Capítulo 2 apresenta a fundamentação teórica sobre os principais conceitos e técnicas necessárias para o entendimento do problema abordado e da solução proposta. O problema tratado neste PFC é descrito em detalhes no Capítulo 3, juntamente com os requisitos técnicos a serem atendidos. No Capítulo 4 é explicada a solução proposta e a metodologia utilizada. O Capítulo 5 aborda o desenvolvimento da solução proposta incluindo as tecnologias utilizadas e a análise dos resultados obtidos. Por fim, no Capítulo 6, são apresentadas as conclusões deste trabalho e algumas sugestões de trabalhos futuros são elencadas.

2 FUNDAMENTAÇÃO TEÓRICA

No capítulo de fundamentação teórica deste trabalho, são abordadas as principais ideias e conceitos que constituem a base teórica para o desenvolvimento da solução proposta. A abordagem metodológica utilizada é pautada em princípios como a inovação tecnológica, a robustez do sistema e a eficiência na gestão de recursos, os quais são fundamentais para resolver problemas específicos enfrentados pelos futuros clientes da Fischertec. A tese apresenta uma nova perspectiva na gestão de reservas de quadras esportivas, com ênfase no uso de aplicativos e serviços web para garantir uma experiência de usuário fluída e personalizável.

2.1 APIS E APIS RESTFUL

APIs (*Application Programming Interfaces*) são interfaces que permitem a comunicação entre diferentes sistemas, softwares ou componentes. Elas funcionam como pontes que facilitam o intercâmbio de dados e funcionalidades, promovendo integração e interoperabilidade. A ideia de API remonta aos primórdios da computação, quando surgiram as primeiras necessidades de criar padrões para que sistemas independentes pudessem se comunicar. Inicialmente, as APIs eram utilizadas em bibliotecas locais para abstrair funcionalidades complexas de hardware e software. Com o avanço da internet, especialmente nas décadas de 1990 e 2000, as APIs evoluíram para um modelo remoto, permitindo que sistemas distribuídos interagissem de maneira mais eficiente.

Nesse contexto, surgiu o conceito de APIs REST, um estilo arquitetural proposto por Roy Fielding em sua tese de doutorado em 2000. REST, acrônimo para *Representational State Transfer*, baseia-se em princípios como uniformidade, ausência de estado (*statelessness*), utilização de métodos HTTP - *Hypertext Transfer Protocol* (GET, POST, PUT, DELETE, etc.), e representação de recursos por meio de URLs (*Universal Resource Locators*). Fielding idealizou REST como uma maneira de padronizar a comunicação entre sistemas na web, tirando proveito das funcionalidades já existentes no protocolo HTTP. APIs REST ganharam popularidade devido à sua simplicidade, flexibilidade e capacidade de escalar, sendo amplamente adotadas por grandes empresas como Google, Twitter e Facebook (Fielding, 2000).

A ligação entre APIs de forma geral e APIs REST está na evolução das necessidades de integração entre sistemas. Enquanto APIs genéricas servem como um conceito abrangente para qualquer forma de comunicação programada entre sistemas, as APIs REST trouxeram um conjunto específico de diretrizes para implementar essas interações de maneira eficiente e alinhada com os padrões da web. Ao adotar REST, as empresas passaram a criar interfaces mais padronizadas e acessíveis, o que facilitou a interoperabilidade entre sistemas desenvolvidos por diferentes organizações.

As vantagens de utilizar APIs, de modo geral, incluem a possibilidade de reuso de funcionalidades, redução de redundâncias no desenvolvimento e facilidade de integração entre diferentes tecnologias. Já as APIs REST se destacam por sua simplicidade e aderência aos padrões da web, tornando-as fáceis de implementar e consumir. Por serem baseadas em HTTP, um protocolo amplamente suportado, as APIs REST são agnósticas em relação à linguagem de programação e oferecem suporte para uma ampla gama de clientes, desde navegadores até dispositivos IoT *(Internet das Coisas)*.

No livro "REST API Design Rulebook", Mark Masse destaca a importância de seguir boas práticas no design de APIs REST: "Uma API RESTful bem projetada deve ter como objetivo fornecer uma interface limpa e intuitiva para que os desenvolvedores interajam com recursos e serviços"(Masse, 2011). Essa ênfase na simplicidade e clareza reflete as razões pelas quais APIs REST têm sido tão amplamente adotadas. Além de facilitar a integração entre sistemas, elas promovem a consistência no design, o que reduz a curva de aprendizado para desenvolvedores e aumenta a produtividade nas equipes de desenvolvimento. Assim, APIs e APIs REST desempenham um papel fundamental no desenvolvimento de soluções modernas e escaláveis, sendo uma base sólida para a inovação tecnológica.

2.2 CLOUD COMPUTING

A computação em nuvem, ou *Cloud Computing*, é um modelo de fornecimento de recursos computacionais, como armazenamento, processamento, bancos de dados e softwares, por meio da internet. A ideia de disponibilizar recursos computacionais de maneira remota surgiu na década de 1960, com o conceito de *time-sharing*, promovido por empresas como IBM e pesquisadores como John McCarthy, que previu que a computação poderia um dia ser organizada como um serviço público. No entanto, a computação em nuvem, como conhecida atualmente, começou a tomar forma nos anos 2000, com o surgimento de grandes provedores de serviços de nuvem, como Amazon Web Services (AWS), Microsoft Azure e Google Cloud Platform (GCP). A AWS, lançada em 2006, é amplamente reconhecida por popularizar a computação em nuvem com serviços como EC2 e S3, que forneceram acesso flexível e escalável a servidores e armazenamento.

Sobre a computação em núvem, Erl, Puttini e Mahmood, afirmam que:

> A computação em nuvem representa uma mudança significativa na forma como os recursos de TI são projetados, implantados e gerenciados, oferecendo um modelo para permitir acesso de rede conveniente e sob demanda a um conjunto compartilhado de recursos de computação configuráveis (Erl; Puttini; Mahmood, 2013).

Na atualidade, a computação em nuvem desempenha um papel essencial no desenvolvimento de software, especialmente no contexto de aplicações web e solu-

ções de Internet das Coisas (IoT). No desenvolvimento de software web, a nuvem permite que empresas de todos os tamanhos utilizem infraestrutura escalável para hospedar seus aplicativos, garantindo alta disponibilidade e desempenho. Por exemplo, plataformas como AWS Elastic Beanstalk e Google App Engine facilitam a implantação e o gerenciamento de aplicativos web, reduzindo a complexidade da configuração de servidores. Além disso, ferramentas de integração e entrega contínuas (CI/CD), como GitHub Actions e Jenkins, são frequentemente integradas com serviços de nuvem, permitindo o desenvolvimento ágil e a atualização frequente de softwares.

No contexto de IoT, a computação em nuvem é fundamental para lidar com o grande volume de dados gerado por dispositivos conectados. Plataformas como AWS IoT Core, Azure IoT Hub e Google Cloud IoT oferecem soluções específicas para conectar, monitorar e gerenciar dispositivos IoT de forma centralizada. Essas plataformas permitem que sensores e dispositivos enviem dados para a nuvem, onde podem ser analisados e utilizados para decisões em tempo real. Por exemplo, em uma aplicação industrial, sensores de máquinas podem enviar informações sobre temperatura e vibração para a nuvem, onde algoritmos de análise preditiva identificam possíveis falhas antes que ocorram. No ambiente residencial, dispositivos como termostatos inteligentes e câmeras de segurança conectadas utilizam serviços de nuvem para armazenamento de dados e controle remoto via aplicativos.

A computação em nuvem oferece vantagens significativas, como escalabilidade sob demanda, custos reduzidos de infraestrutura e maior flexibilidade para equipes de desenvolvimento. Para projetos de software, ela possibilita o processamento de grandes volumes de dados, integração de dispositivos globais e implantação rápida de novas funcionalidades. Com a crescente demanda por soluções conectadas e inovadoras, a nuvem continua a se consolidar como uma tecnologia essencial para o avanço da computação moderna.

2.3 SOFTWARE AS A SERVICE (SAAS)

Software como Serviço (SaaS, do inglês *Software as a Service*) é um modelo de distribuição de software baseado na nuvem, em que as aplicações são hospedadas por um provedor e acessadas pelos usuários por meio da internet, geralmente por meio de um navegador. Essa abordagem elimina a necessidade de instalação e manutenção de softwares locais, proporcionando uma experiência mais acessível e simplificada (Turner; Budgen; Brereton, 2003). A origem do conceito pode ser rastreada até a década de 1960, com a introdução do *time-sharing*, um modelo de computação em que múltiplos usuários podiam acessar sistemas centrais compartilhados. Entretanto, o modelo SaaS começou a ganhar relevância com o avanço da computação em nuvem nos anos 2000, especialmente após a popularização de empresas como Salesforce, que lançou sua plataforma de CRM (*Customer Relationship Management*) baseada na

web em 1999, marcando um marco importante para o setor.

Na atualidade, o SaaS é amplamente utilizado em diversos setores, oferecendo soluções que vão desde produtividade e colaboração até análises de dados e gestão empresarial. Exemplos populares incluem ferramentas como o Google Workspace, que oferece aplicativos como Google Docs, Google Sheets e Google Drive, acessíveis diretamente no navegador e armazenados na nuvem. Plataformas como Slack e Microsoft Teams são usadas para comunicação e colaboração em equipe, eliminando a necessidade de servidores locais de e-mail ou mensagens instantâneas. Softwares de gestão empresarial, como SAP e NetSuite, também utilizam o modelo SaaS para oferecer recursos avançados de ERP (*Enterprise Resource Planning*) de forma escalável e acessível.

No setor de entretenimento, plataformas como Netflix e Spotify exemplificam o modelo SaaS ao oferecerem serviços de streaming de vídeo e música, respectivamente, sem a necessidade de que o conteúdo seja baixado ou armazenado localmente. Na área da educação, o SaaS tem permitido o crescimento de plataformas de aprendizado online, como Coursera e Khan Academy, que fornecem acesso a cursos e materiais educacionais diretamente pela web. Até mesmo áreas como saúde e finanças adotaram soluções SaaS, com softwares que ajudam na gestão de pacientes ou no controle financeiro de empresas.

As vantagens do SaaS incluem custos iniciais mais baixos, escalabilidade, acessibilidade de qualquer lugar com conexão à internet e atualizações automáticas realizadas pelo provedor. Além disso, o modelo permite que as empresas se concentrem em suas atividades principais, sem a necessidade de gerenciar infraestrutura de TI complexa. Embora existam desafios relacionados à segurança de dados e à dependência de conectividade, o SaaS continua a ganhar popularidade devido à sua flexibilidade e capacidade de atender às demandas de um mercado global em constante evolução.

2.4 CONTEINERIZAÇÃO

A containerização é uma tecnologia que revolucionou a maneira como aplicações são desenvolvidas, implantadas e gerenciadas, promovendo maior eficiência e consistência no ambiente de execução. Sua história remonta à década de 1970, com conceitos iniciais relacionados à virtualização e à criação de ambientes isolados em sistemas Unix, como o *chroot*. No entanto, foi apenas na década de 2000 que a containerização começou a ganhar tração significativa, especialmente com o surgimento de ferramentas como o Linux Containers (LXC) e, mais tarde, o Docker em 2013. O Docker popularizou a tecnologia ao torná-la acessível, padronizada e mais eficiente, contribuindo para sua ampla adoção em ambientes de desenvolvimento e produção.

A principal função da containerização é fornecer um ambiente isolado para aplicações, encapsulando todo o necessário para sua execução, como código, bibliotecas

e dependências, em um único contêiner. Esses contêineres são leves, portáteis e consistentes, podendo ser executados em qualquer ambiente que suporte a tecnologia, como servidores locais, nuvens públicas ou privadas. Atualmente, a containerização é amplamente utilizada no desenvolvimento de software, especialmente em arquiteturas de microsserviços, onde cada serviço é executado em seu próprio contêiner, facilitando a escalabilidade e a manutenção. Além disso, é empregada em projetos de Internet das Coisas (IoT), onde dispositivos podem executar contêineres específicos para gerenciar funcionalidades de forma modular.

Entre as tecnologias que utilizam a containerização, destacam-se o Docker e o Kubernetes. O Docker é uma plataforma que permite criar, gerenciar e executar contêineres de maneira simplificada, enquanto o Kubernetes é um sistema de orquestração que automatiza o gerenciamento de contêineres em larga escala, garantindo alta disponibilidade, balanceamento de carga e escalabilidade automática. Essas ferramentas são amplamente adotadas em empresas de todos os tamanhos para aprimorar fluxos de trabalho de DevOps, modernizar aplicações legadas e facilitar a entrega contínua de software.

A containerização trouxe diversas vantagens em relação às abordagens tradicionais de virtualização, como menor sobrecarga, maior eficiência no uso de recursos e tempos de inicialização mais rápidos. Além disso, ao garantir que as aplicações sejam executadas de forma consistente em diferentes ambientes, a tecnologia reduz problemas de compatibilidade e acelera os ciclos de desenvolvimento e implantação. Com sua versatilidade e eficiência, a containerização continua a transformar a forma como o software é projetado, implementado e executado em diversos setores.

2.5 ARQUITETURA EM CAMADAS

A arquitetura em camadas é um modelo amplamente adotado no desenvolvimento de aplicações modernas, proporcionando uma estrutura clara e organizada que facilita a manutenção, escalabilidade e reutilização de código. Esta abordagem divide a aplicação em camadas distintas, cada uma com responsabilidades específicas e bem definidas. No desenvolvimento da aplicação, a arquitetura em camadas é utilizada para separar as preocupações, garantindo que cada componente funcione de forma independente e coesa. (Fowler, 2002) descreve a arquitetura em camadas como uma abordagem fundamental para a construção de aplicações corporativas, destacando sua popularidade e uso extensivo em aplicações empresariais. Cada uma das camadas é apresentada da Seção 2.5.1 à Seção 2.5.5.

2.5.1 Camada de Apresentação

A camada de apresentação é responsável pela interface com o usuário final e pode ser desenvolvida utilizando diversas tecnologias e frameworks, como Next.js com a biblioteca React.js, Vue.js, Angular.js, HTML e CSS entre outros. O Next.js, por exemplo, com sua capacidade de renderização híbrida (SSR e SSG), oferece uma experiência de usuário rápida e otimizada para SEO, enquanto o React possibilita a criação de componentes reutilizáveis e uma interface dinâmica e responsiva. Nesta camada, as interações do usuário são capturadas e encaminhadas para a camada de aplicação geralmente através de chamadas HTTP a APIs RESTful, garantindo uma separação clara entre a interface e a lógica de negócios, porém outros protocolos podem ser usados para a comunicação.

2.5.2 Camada de Aplicação

A camada de aplicação atua como intermediária entre a interface do usuário e a lógica de negócios. Executando no servidor, esta camada gerencia a lógica de controle e orquestra as operações entre a camada de apresentação e a camada de negócios. Diversas tecnologias podem ser usadas para o desenvolvimento da aplicação tais como Node.js, Nest.js, Java, Spring, C#, .NET, PHP, Python entre outros. Nesta camada, são definidos controladores que lidam com as requisições recebidas, e as enchaminham para a camada de negócios adequada para o processamento dos dados.

2.5.3 Camada de Negócios

A camada de negócios encapsula a lógica principal do sistema, garantindo que as regras de negócios sejam aplicadas de forma consistente. Esta camada é responsável pela implementação das regras de autenticação e autorização de usuários, além de toda a lógica necessária para o funcionamento adequado da aplicação. Ao concentrar as regras de negócio nesta camada, a aplicação assegura que todas as operações críticas são tratadas de forma centralizada e independente das outras camadas, simplificando o desenvolvimento, manutenção e melhoria do código da aplicação.

2.5.4 Camada de Persistência

A camada de persistência é responsável por gerenciar a interação com o banco de dados de forma abstrata ou direta. Pode-se utilizar tecnologias como TypeORM e Prisma nesta camada para abstrair a comunicação ou usar diretamente as *queries* para interação com o banco de dados, permitindo realizar operações de criação, leitura, atualização e exclusão de dados de forma simplificada e eficiente.

2.5.5 Camada de Banco de Dados

A camada de banco de dados é o próprio sistema de armazenameto escolhido para o projeto, podendo ser um banco de dados estruturado (como PostgreSQL, MySQL, Oracle entre outro) ou não (como MongoDB, Cassandran InfluxDB, etc). Esta camada é responsável por armazenar e recuperar os dados persistidos pela camada de persistência. A escolha adequada de um sistema de banco de dados garante a escalabilidade, consistência e disponibilidade dos dados, além de permitir o uso de recursos avançados para otimizar o desempenho.

2.5.6 Benefícios da Arquitetura em Camadas

A arquitetura em camadas traz vários benefícios para o desenvolvimento desta aplicação. A separação de preocupações facilita a manutenção do código, permitindo que alterações em uma camada específica não afetem diretamente as demais. Isso também melhora a escalabilidade da aplicação, pois novas funcionalidades podem ser adicionadas ou modificadas de forma isolada. Além disso, a modularização contribui para uma melhor reutilização de componentes, tornando o desenvolvimento mais eficiente e ágil. (Martin, Robert C., 2017) discute como a arquitetura em camadas permite uma divisão clara de responsabilidades, promovendo independência no desenvolvimento e manutenção.

Portanto a adoção de uma arquitetura em camadas proporciona uma estrutura organizada e clara, que facilita o desenvolvimento, manutenção e evolução da aplicação. Cada camada desempenha um papel essencial, garantindo que a aplicação seja robusta, escalável e fácil de manter. A utilização desta abordagem, aliada às tecnologias escolhidas, assegura a entrega de um sistema eficiente, confiável e alinhado às melhores práticas do desenvolvimento de software moderno.

2.6 ARQUITETURA MODULAR

A arquitetura modular, também conhecida como arquitetura em módulos, é um estilo de design de software que organiza o sistema em componentes independentes, chamados módulos. Cada módulo encapsula uma funcionalidade específica e é projetado para ser autônomo, permitindo que diferentes partes do sistema possam ser desenvolvidas, testadas, mantidas e implantadas de forma independente. Esse tipo de arquitetura promove a separação de responsabilidades, facilitando o trabalho colaborativo entre equipes e reduzindo a complexidade no gerenciamento do sistema como um todo.

Os módulos comunicam-se entre si através de interfaces bem definidas, garantindo que as dependências entre eles sejam minimizadas e bem gerenciadas. A modularidade não apenas melhora a organização do código, mas também torna o

sistema mais escalável, permitindo que novos recursos sejam adicionados sem comprometer as funcionalidades existentes.

Alguns dos principais benefícios da arquitetura modular são: a reusabilidade do código, já que um módulo pode ser utilizado em diferentes contextos; a facilidade na adição ou remoção de funcionalidades; a escalabilidade, simplificando a expansão do sistema; a capacidade de gerenciar alterações independentemente, além de facilitar a manutenção e desenvolvimento paralelo do código em diferentes times ou projetos.

2.7 INVERSÃO E INJEÇÃO DE DEPENDÊNCIAS

A inversão de dependências é um dos princípios fundamentais do desenvolvimento de software orientado a objetos, sendo parte dos cinco princípios do SOLID, apresentados pela primeira vez por Robert C. Martin. Esse conceito está diretamente relacionado à ideia de desacoplar componentes de software, de modo a tornar o código mais flexível, reutilizável e de fácil manutenção. No artigo **Design Principles and Design Patterns**, Martin descreve o **Princípio de Inversão de Dependências (DIP - *Dependency Injection Principle*)** como a prática de depender de abstrações, e não de implementações concretas (Martin, Robert C, 2000, p. 12). Esse princípio defende que módulos de alto nível não devem depender de módulos de baixo nível diretamente; ambos devem depender de abstrações, como interfaces ou classes abstratas. Essa abordagem reduz o impacto de mudanças no sistema, pois a implementação concreta pode ser alterada sem afetar os módulos que a utilizam.

A injeção de dependências, por sua vez, é uma técnica para implementar o princípio de inversão de dependências. Ela consiste em fornecer as dependências que uma classe precisa a partir de fora, em vez de a própria classe criar ou localizar essas dependências. Em outras palavras, a responsabilidade de instanciar ou gerenciar dependências é transferida para um container ou outro componente externo. Existem três formas principais de realizar a injeção de dependências: por meio de construtores, métodos ou atributos da classe. Essa prática é amplamente adotada por frameworks modernos, como o Spring no Java e o NestJS no TypeScript, que oferecem containers de injeção de dependências para gerenciar automaticamente a criação e a injeção de objetos.

A ligação entre o princípio de inversão de dependências e a injeção de dependências é clara: a injeção de dependências é uma maneira prática de aplicar o DIP, permitindo que as classes dependam de abstrações em vez de implementações concretas. Por exemplo, em uma aplicação que utiliza um repositório para acessar o banco de dados, o repositório deve ser definido como uma interface (abstração), e a implementação concreta desse repositório deve ser fornecida à classe dependente por meio de injeção de dependências. Isso promove a substituição fácil da implementação do repositório, caso seja necessário, como ao trocar um banco de dados relacional por

um banco NoSQL - *Not Only Structured Query Language*.

As vantagens de adotar o princípio de inversão de dependências e a injeção de dependências são muitas. Primeiramente, essas práticas reduzem o acoplamento entre os componentes do sistema, tornando o código mais modular e fácil de manter. A testabilidade também é significativamente melhorada, já que as dependências podem ser facilmente substituídas por objetos simulados (*mocks*) durante a execução de testes. Além disso, a flexibilidade do sistema aumenta, pois mudanças em uma implementação concreta têm impacto limitado nas demais partes do sistema. Essas práticas promovem também a reutilização de código, já que módulos de alto nível não são diretamente ligados às implementações de baixo nível.

Como afirma Robert C. Martin em seu artigo, módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações (Martin, Robert C, 2000). Esse princípio continua sendo um dos pilares para o desenvolvimento de software robusto e escalável, demonstrando sua relevância mesmo em arquiteturas modernas e frameworks contemporâneos.

O princípio da inversão de dependência é uma fundamental característica da arquitetura modular, que busca reduzir a dependência direta entre componentes e promover a independência das camadas. Como essa abordagem sugere que as classes ou módulos devem depender de abstrações gerais em vez de específicas, é possível alterar uma implementação sem afetar outras partes do sistema, tornando a manutenção e evolução mais fáceis.

2.8 IOT

A Internet das Coisas (IoT, do inglês *Internet of Things*) é um conceito que se refere à interconexão de dispositivos físicos com a internet, permitindo que eles coletem, compartilhem e atuem com base em dados, promovendo automação e eficiência em diversos setores (Gokhale; Bhat, O.; Bhat, S., 2018). A ideia de conectar dispositivos remonta à década de 1980, quando surgiu o conceito de "computação ubíqua", proposto por Mark Weiser, que vislumbrava a tecnologia como parte integral do cotidiano, funcionando de maneira invisível aos usuários. Entretanto, a IoT começou a ganhar forma prática nos anos 1990 com a popularização da internet e dos avanços em sensores, processadores e redes sem fio. Um marco importante foi a introdução de dispositivos como a máquina de venda de refrigerantes da Coca-Cola, em 1982, considerada um dos primeiros dispositivos "inteligentes", capaz de reportar seu status em tempo real.

Na atualidade, a IoT está presente em diversos setores, tanto na indústria quanto em residências e cidades. No setor industrial, a IoT é aplicada na automação de processos produtivos, monitoramento de máquinas e manutenção preditiva, configurando o que é chamado de Indústria 4.0. Sensores conectados podem, por

exemplo, monitorar condições de equipamentos, como temperatura, vibração ou consumo energético, alertando sobre possíveis falhas antes que elas ocorram. Isso reduz custos operacionais e aumenta a eficiência. Além disso, a IoT é usada na logística, rastreando mercadorias em tempo real e otimizando rotas de transporte.

Em ambientes residenciais, a IoT está transformando as casas em espaços inteligentes. Dispositivos como termostatos conectados, lâmpadas controladas por aplicativos, assistentes virtuais (como Alexa ou Google Assistant) e câmeras de segurança inteligentes são exemplos de como a IoT está sendo integrada ao dia a dia das pessoas. Esses dispositivos permitem automação, controle remoto e até mesmo o uso de inteligência artificial para aprendizado dos hábitos dos moradores, aumentando conforto, segurança e eficiência energética.

Nas cidades, a IoT tem papel fundamental na construção de "cidades inteligentes". Exemplos incluem semáforos que ajustam automaticamente seus ciclos de acordo com o fluxo de tráfego, sistemas de coleta de lixo que otimizam rotas de caminhões com base no nível de preenchimento das lixeiras e sensores ambientais que monitoram a qualidade do ar em tempo real. Esses avanços visam melhorar a qualidade de vida dos cidadãos, reduzindo custos e impactos ambientais.

A IoT também é amplamente utilizada em setores como saúde, agricultura e varejo. No campo da saúde, dispositivos vestíveis como relógios inteligentes monitoram sinais vitais e enviam alertas em casos de anomalias. Na agricultura, sensores conectados ajudam no monitoramento do solo, otimizando o uso de água e fertilizantes. No varejo, prateleiras inteligentes e etiquetas RFID (*Radio Frequency Identification*) permitem rastrear estoques em tempo real e melhorar a experiência do cliente.

A IoT está revolucionando a forma como as pessoas interagem com o mundo ao seu redor, tornando processos mais inteligentes e conectados. Embora ainda enfrente desafios relacionados à segurança, privacidade e interoperabilidade, seu impacto já é visível em diversos aspectos da sociedade moderna, com um potencial imenso de expansão nos próximos anos.

3 DESCRIÇÃO DO PROBLEMA E REQUISITOS TÉCNICOS

Ao longo deste capítulo, será abordado a contextualização dos desafios enfrentados pelas empresas de aluguel de quadras esportivas, a descrição do problema tratado no PFC, os requisitos técnicos a serem atendidos, além da solução proposta pelo autor para o problema.

3.1 CONTEXTUALIZAÇÃO

As empresas que operam no setor de aluguel de quadras esportivas enfrentam uma série de desafios relacionados à gestão de reservas, manutenção das instalações e otimização de recursos. Em um mercado cada vez mais competitivo, a eficiência na administração dos espaços torna-se um diferencial essencial para a satisfação dos clientes e a sustentabilidade financeira do empreendimento.

Um dos principais desafios enfrentados é a necessidade de um sistema confiável e automatizado para o agendamento de reservas das quadras. Muitos estabelecimentos ainda utilizam processos manuais, como anotações em papel ou planilhas eletrônicas, o que aumenta o risco de erros, como agendamentos duplicados ou conflitos de horários. Essa falta de automação também dificulta a visualização em tempo real da disponibilidade das quadras, tornando a experiência do cliente menos eficiente e satisfatória.

Manter os clientes informados e atualizados quanto aos horários disponíveis para cada quadra é outro desafio significativo. Sem um sistema centralizado e integrado, os usuários precisam entrar em contato diretamente com a administração do estabelecimento para verificar a disponibilidade, o que consome tempo e pode resultar na perda de reservas devido à demora na resposta. A falta de uma plataforma acessível que permita consultas e reservas instantâneas compromete a conveniência e a eficiência operacional do negócio.

Outro aspecto crítico é o desperdício de recursos energéticos e humanos na gestão das quadras. A iluminação e os sistemas de climatização das quadras precisam ser ligados e desligados de acordo com o uso, mas, muitas vezes, essa tarefa é realizada manualmente pelos funcionários, resultando em inconsistências, desperdício de energia e custos operacionais elevados. Quadras que permanecem iluminadas ou climatizadas mesmo sem uso impactam diretamente nas despesas mensais da empresa e no consumo desnecessário de energia, causando impacto ambiental negativo.

Diante desses desafios, a automação e digitalização do processo de gerenciamento de quadras esportivas tornam-se fundamentais para garantir maior eficiência, redução de custos operacionais e melhor experiência do usuário. A adoção de um sistema integrado para reservas, comunicação com os clientes e controle de dispositivos IoT apresenta-se como a solução mais eficaz para modernizar o setor e tornar a

gestão mais inteligente e automatizada.

3.2 DESCRIÇÃO DO PROBLEMA

Com base nos desafios apresentados, identifica-se a necessidade de um sistema capaz de otimizar a gestão de reservas de quadras esportivas, garantindo maior controle sobre a disponibilidade dos espaços e facilitando a comunicação com os clientes. A ausência de uma plataforma centralizada impede que os usuários realizem reservas de maneira prática e intuitiva, resultando em processos administrativos demorados e suscetíveis a erros. Além disso, a falta de automação nos processos operacionais aumenta os custos e impacta negativamente na sustentabilidade do negócio.

A inexistência de um controle eficiente sobre o uso de iluminação e climatização das quadras contribui para o desperdício de recursos, elevando os custos operacionais e causando impacto ambiental. A dependência de funcionários para gerenciar manualmente esses dispositivos aumenta a possibilidade de falhas, como equipamentos ligados sem necessidade ou desligados em momentos inadequados. Dessa forma, a implementação de uma solução tecnológica que integre a gestão de reservas e o controle de dispositivos IoT torna-se essencial para melhorar a eficiência operacional e reduzir desperdícios.

Portanto, a ausência de uma solução integrada e automatizada afeta diretamente a operação das empresas que alugam quadras esportivas, limitando seu crescimento e impactando a experiência do usuário. A implementação de um sistema completo e eficiente, que contemple reservas online e controle automatizado de dispositivos IoT, surge como a resposta para superar esses desafios e proporcionar uma administração mais moderna e eficaz.

3.3 REQUISITOS TÉCNICOS A SEREM ATENDIDOS

O desenvolvimento de uma solução para esses problemas exige a definição clara de requisitos técnicos que devem ser atendidos para garantir a entrega de uma solução eficiente, robusta e alinhada às necessidades do negócio. Estes requisitos estão divididos em funcionais dispostos na Seção 3.3.1 e não-funcionais na Seção 3.3.2, cobrindo tanto as funcionalidades esperadas quanto os atributos de qualidade do sistema. Além disso, alguns casos de uso ilustram as principais interações previstas na aplicação na Seção 3.3.3. Como o foco deste PFC é a implementação do servidor, os requisitos técnicos estão mais voltados para a arquitetura e as funcionalidades do sistema, deixando a interface de usuário para um desenvolvimento futuro.

3.3.1 Requisitos Funcionais

1. O sistema deve permitir o cadastro de empresas no formato multi-tenant, onde cada empresa terá um ambiente isolado para suas configurações e dados.
2. Deve ser possível configurar os horários de funcionamento das quadras de forma personalizada para cada dia da semana.
3. O sistema deve oferecer uma interface para visualizar os horários configurados de uma quadra para todos os dias da semana.
4. Deve ser possível realizar reservas de quadras, com confirmação imediata e registro dos dados no sistema.
5. O administrador deve poder cadastrar, editar e excluir quadras do sistema.
6. A integração com a API eWeLink deve permitir o controle de dispositivos IoT, como iluminação e climatização, associados às quadras.
7. Deve ser possível configurar comandos automáticos para ligar e desligar dispositivos IoT no início e no final das reservas.
8. Deve haver suporte para diferentes níveis de acesso, como administradores, funcionários e clientes.
9. Um usuário deve poder ter diferentes papéis em empresas distintas.
10. O sistema deve ser capaz de ignorar comandos automáticos para o mesmo dispositivo em caso de reservas seguidas, evitando ligações e desligamentos desnecessários.

3.3.2 Requisitos Não-Funcionais

- O sistema deve ser desenvolvido como uma aplicação web baseada em cloud, garantindo disponibilidade e escalabilidade.
- O aplicação deve ser desenvolvida em uma linguagem consolidada no mercado e com tipagem de dados, afim de simplificar futuras manutenções e melhorias.
- O banco de dados utilizado deve ser relacional, garantindo consistência e integridade dos dados.
- A aplicação deve suportar múltiplos usuários simultâneos sem comprometimento de desempenho.
- Deve ser garantida a segurança dos dados por meio de autenticação e autorização robustas.

- O administrador da empresa deve ter a possibilidade de vincular sua conta eWelink de forma prática.
- O administrador da empresa deve ser capaz de adicionar novos dispositivos IoT mesmo sem ter conhecimento técnico avançado.
- O sistema deve manter logs de eventos críticos, como reservas realizadas e comandos enviados a dispositivos IoT.
- O tempo de resposta das principais operações deve ser inferior a 2 segundos.
- A aplicação deve ser documentada para facilitar a manutenção e futuras expansões.

3.3.3 Casos de Uso

Caso de Uso 1: Reserva de Quadra

- Ator: Cliente
- Descrição: O cliente acessa o sistema através de um navegador, visualiza as quadras disponíveis para uma data e horário específicos, seleciona a quadra desejada e realiza a reserva. O sistema confirma a reserva e cria os comandos para ligar e desligar as luzes da quadra no início e fim da reserva.

Caso de Uso 2: Configuração de Quadras e Dispositivos IoT

- Ator: Administrador
- Descrição: O usuário faz login na plataforma e cria uma nova empresa. Esse usuário se torna administrador automaticamente para essa empresa. Ele acessa o painel de administração da empresa, adiciona uma nova quadra e associa um dispositivo IoT e um comando de Ïigarão início dos agendamentos dessa quadra.

Caso de Uso 3: Consulta de Disponibilidade de Quadras

- Ator: Cliente
- Descrição: O cliente acessa a aplicação, seleciona uma empresa e uma consulta a disponibilidade de uma quadra para um dia específico. As informações da disponibilidade do dia da semana para a quadra selecionada são exibidas. Os horários indisponíveis devem ser apresentados a partir dos agendametos confirmados nessa quadra.

Caso de Uso 4: Cancelamento de Reserva de Quadra

- Ator: Cliente
- Descrição: O cliente acessa a aplicação e escolhe ver as reservas que tem para uma determinada empresa. O sistema busca todas as quadras desse empresa e busca as reservas do cliente para cada quadra. O cliente seleciona a reserva que deseja cancelar e o sistema confirma o cancelamento, liberando a quadra para novas reservas.

Com esses requisitos e casos de uso bem definidos, a aplicação está preparada para atender às necessidades do mercado, oferecendo uma solução moderna, eficiente e alinhada às expectativas dos usuários e administradores.

4 SOLUÇÃO PROPOSTA E METODOLOGIA UTILIZADA

Este capítulo tratará da solução proposta e metodologia utilizada, bem como soluções alternativas que foram consideradas e descartadas ao longo do processo de proposta de uma solução para o problema considerado.

4.1 SOLUÇÃO PROPOSTA

4.1.1 Visão Geral

Para atender aos desafios identificados no Capítulo 3, propõe-se o desenvolvimento de um sistema web baseado em *cloud computing*, com uma abordagem de *Software as a Service* (SaaS), voltado para o gerenciamento de reservas de quadras esportivas e integração com dispositivos IoT. Uma visão geral da arquitetura do sistema está demonstrada na Figura 1. Esse sistema será composto por um backend responsável por toda a lógica de negócios e armazenamento dos dados, e um frontend voltado para a interação com os clientes e a administração das quadras, o qual será desenvolvido em um projeto futuro, além de um banco de dados para armazenar informações da aplicação.

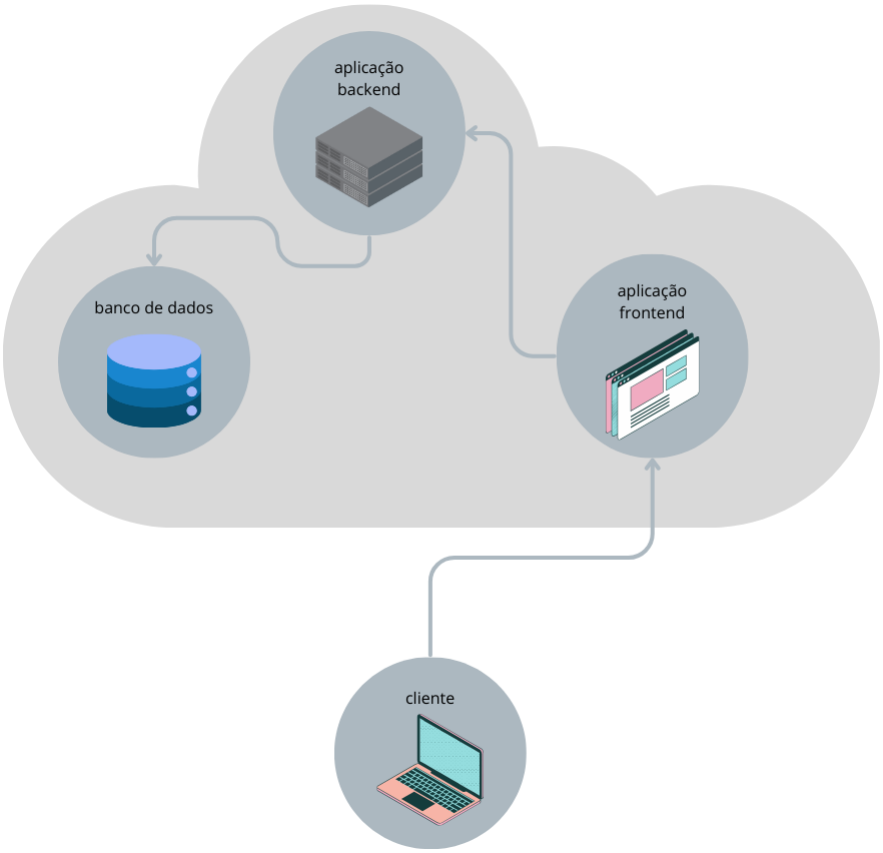
O sistema adotará uma abordagem *multi-tenant*, permitindo que múltiplas empresas utilizem a mesma plataforma de maneira independente. O administrador de cada empresa poderá configurar seu próprio ambiente, definindo nome, horários de funcionamento e quantidade de quadras disponíveis. Cada quadra terá horários de funcionamento ajustáveis de acordo com os dias da semana, garantindo flexibilidade na gestão.

Também foi considerada uma abordagem *single-tenant* e replicável para cada cliente. O problema dessa abordagem é que a manutenção de cada instância do sistema seria mais custosa e complexa, pois cada cliente teria seu próprio banco de dados e instância do sistema. Além disso a necessidade de configuração manual para cada novo cliente seria trabalhosa e custosa para o time da Fischertec. A abordagem *multi-tenant* é mais escalável e econômica, pois todos os clientes compartilham a mesma instância do sistema e banco de dados, com isolamento de dados e configurações.

Os clientes esportistas poderão acessar a plataforma para visualizar a disponibilidade das quadras de uma empresa em tempo real, selecionar um horário específico e realizar a reserva de forma simples e intuitiva. A confirmação das reservas será imediata por padrão, proporcionando uma experiência eficiente e conveniente tanto para os usuários quanto para os administradores. Será possível, também, configurar a necessidade de aprovação para as reservas, caso a empresa deseje revisar manualmente cada solicitação.

A fim de garantir a integração com dispositivos IoT de maneira simples e sem

Figura 1 – Arquitetura geral da aplicação.



Fonte: Autor.

necessidade de conhecimento técnico por parte dos administradores das empresas, optou-se por utilizar uma solução de hardware consolidada no mercado, a Sonoff, que oferece dispositivos de automação residencial com suporte a integração via API. Dessa maneira, evita-se a necessidade de um hardware e uma aplicação extra que se conectaria ao servidor fazendo uma espécie de ponte para gerenciar os dispositivos IoT diretamente através da rede local. Além disso seria necessário uma configuração extra para liberar o controle local nos dispositivos Sonoff (o modo DIY *Do It Yourself*) e atribuição de ips fixos na rede local (pois em caso de mudança de ip, seria necessário reconfigurar a aplicação de ponte para que ela pudesse encontrar os dispositivos na rede).

Portanto, a automação do controle dos dispositivos IoT será feita por meio da integração com a plataforma eWeLink da Shenzhen Coolkit Technology CO., LTD, amplamente utilizada no mercado e compatível com dispositivos da marca Sonoff. O administrador poderá conectar sua conta eWeLink ao sistema, permitindo a identificação automática dos dispositivos configurados. Esses dispositivos poderão ser atribuídos

às quadras e configurados para executar comandos específicos no início e/ou fim de uma reserva, como ligar ou desligar iluminação e climatização de forma automatizada. Essa solução permitirá que os administradores da empresa configurem facilmente os dispositivos IoT para funcionarem em conjunto com a plataforma, sem a necessidade de contatar profissionais de tecnologia ou executar instruções complexas, reduzindo a abertura de chamados de suporte da aplicação.

Com essa abordagem, o sistema proporcionará um gerenciamento eficiente e automatizado das quadras esportivas, reduzindo desperdícios, otimizando recursos e melhorando a experiência dos clientes. Além disso, a solução contribuirá para a redução de custos operacionais e impacto ambiental, tornando-se uma ferramenta indispensável para empresas que buscam modernização e eficiência em sua gestão.

O desenvolvimento do sistema completo será dividido em duas etapas: backend e frontend. Neste projeto de fim de curso, será abordada a implementação do backend, que consiste na construção de uma aplicação de servidor responsável por toda a lógica de negócios e integração com a plataforma eWeLink. A aplicação será desenvolvida em Node.js, utilizando o framework Nest.js na linguagem Typescript, afim de satisfazer os requisitos técnicos apresentados. Terá suporte a cadastro, autenticação e autorização de usuários, persistência de dados em um banco de dados PostgreSQL e integração com a API da eWeLink.

5 DESENVOLVIMENTO E ANÁLISE DOS RESULTADOS OBTIDOS

Este capítulo apresenta o processo de desenvolvimento do projeto de fim de curso relativo ao desenvolvimento de um sistema de gerenciamento de reservas e controle automatizado de dispositivos IoT para quadras esportivas. Na Seção 5.1 são apresentadas as ferramentas e tecnologias utilizadas no desenvolvimento do sistema. A Seção 5.2 apresenta o processo de desenvolvimento do PFC e a Seção 5.3 dispõe dos resultados alcançados ao fim do processo.

Alguns pontos que serão explicados nesse capítulo:

- **Ferramentas**: As tecnologias utilizadas na implementação da solução proposta, como linguagens de programação, frameworks, bibliotecas, etc.
- **Arquitetura da solução**: A arquitetura utilizada na implementação da solução proposta, como monolítica, microsserviços, cliente-servidor, etc.
- **Modelagem do banco de dados**: O modelo utilizado para representar os dados da aplicação, incluindo as entidades, relacionamentos, atributos, etc.
- **Backend**: A parte da solução que lida com a manipulação dos dados e a regra de negócio da aplicação.

  Registro e autenticação de usuários: Como o backend implementa a funcionalidade de registro e autenticação de usuários, incluindo validações, segurança, etc.

  Autorização de usuários: Como o backend implementa a funcionalidade de autorização de usuários, incluindo permissões, roles, etc.

  Rotas da aplicação: Como o backend implementa as rotas da aplicação, incluindo endpoints de CRUD para tenanants, quadras, agendamentos de horários e credenciais de acesso a API relacionada à internet das coisas (IoT).

  Detalhamento do agendamentos de horários: Como o backend implementa a funcionalidade de agendamentos de horários de quadras, incluindo informações sobre os locais, os tipos de atividades, as datas e as horas.

  Detalhamento da integração com IoT: Como o backend implementa a integração com dispositivos IoT e comanda o acionamento e desligamento de acordo com os agendamentos feitos pelosus usuários.
- **Análise de resultados**: Análise dos resultados obtidos com o desenvolvimento da aplicação. Detalhando os principais aspesctos relevantes, como a eficiência do agendamento de horários, a segurança das informações e a integração com IoT.

5.1 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

A seção de ferramentas e tecnologias utilizadas no desenvolvimento deste projeto apresenta um panorama detalhado dos recursos técnicos empregados para a construção da aplicação de servidor do sistema de gerenciamento de reservas e controle automatizado de dispositivos IoT para quadras esportivas.

5.1.1 Git

O Git é um sistema de controle de versão distribuído criado por Linus Torvalds em 2005, inicialmente para o desenvolvimento do kernel do Linux. Antes do Git, o projeto Linux utilizava o BitKeeper, mas devido a restrições de licenciamento, Torvalds decidiu desenvolver uma ferramenta própria. O objetivo principal era criar um sistema rápido, eficiente e que suportasse grandes projetos com facilidade.

As principais funções do Git incluem rastreamento de alterações no código-fonte, permitindo que múltiplos desenvolvedores trabalhem simultaneamente em um mesmo projeto sem conflitos. O Git possibilita a criação de ramificações (branches) para desenvolver novos recursos ou corrigir bugs de forma isolada, facilitando a integração dessas alterações ao projeto principal posteriormente. Além disso, como um sistema distribuído, cada desenvolvedor possui uma cópia completa do repositório, o que aumenta a segurança e a integridade dos dados. Atualmente, o Git é a ferramenta de controle de versão mais popular do mundo, amplamente utilizada por empresas de software, desenvolvedores independentes e projetos de código aberto.

Decidiu-se utilizar o Git devido a familiaridade da equipe com a ferramenta, já que é amplamente usada em outros projetos na empresa, além de sua robustez e confiabilidade.

O Git revolucionou a forma como o desenvolvimento de software é conduzido, proporcionando uma maneira eficiente e colaborativa de gerenciar projetos complexos. Sua flexibilidade, velocidade e robustez tornaram-no uma ferramenta essencial no arsenal de qualquer desenvolvedor, suportando desde pequenos projetos individuais até grandes sistemas empresariais.

5.1.2 GitHub

O GitHub é uma plataforma de hospedagem e repositório de código-fonte e arquivos que utiliza o Git para controle de versão. A plataforma permite que programadores ou qualquer usuário cadastrado contribuam em projetos, sejam eles privados ou de código aberto, de qualquer lugar do mundo. Amplamente adotada por desenvolvedores, o GitHub facilita a divulgação de trabalhos e a colaboração em projetos, além de oferecer recursos que simplificam a comunicação, como a identificação de problemas e a mesclagem de repositórios remotos por meio de issues e pull requests.

Atualmente, o GitHub é utilizado globalmente, contando com mais de 36 milhões de usuários ativos que contribuem em projetos comerciais ou pessoais. A plataforma hospeda mais de 100 milhões de projetos, incluindo alguns de renome mundial, como WordPress, GNU/Linux, Atom e Electron. Além disso, o GitHub oferece suporte a recursos de organização, amplamente utilizados por aqueles que desejam expandir seus projetos.

O Github é uma ferramenta usada por padrão na Fischertec. Embora outras plataformas como o Gitlab tenham funções semelhantes, optou-se por manter o padrão da empresa.

5.1.3 PostgreSQL

O SQL (Structured Query Language) foi desenvolvido nos anos 1970 pela IBM como uma linguagem padrão para gerenciar e manipular bancos de dados relacionais. PostgreSQL, um sistema de gerenciamento de banco de dados relacional, surgiu na Universidade da Califórnia em Berkeley, no final dos anos 1980, como parte do projeto POSTGRES (Post Ingres). Em 1996, o sistema foi renomeado para PostgreSQL, refletindo sua compatibilidade com SQL. Desde então, PostgreSQL evoluiu significativamente, tornando-se uma das bases de dados relacionais mais robustas e avançadas disponíveis.

PostgreSQL oferece uma ampla gama de funções, incluindo suporte a transações ACID (Atomicidade, Consistência, Isolamento, Durabilidade), integridade referencial, e extensões para manipulação de dados complexos como JSON e XML. Além disso, suporta procedimentos armazenados, gatilhos e uma variedade de tipos de índices para otimizar consultas. Atualmente, PostgreSQL é amplamente utilizado por grandes empresas, startups e projetos de código aberto devido à sua confiabilidade, flexibilidade e conformidade com os padrões SQL.

Existem outras soluções de banco de dados SQL como MySQL, que é conhecido por sua facilidade de uso e performance em aplicações web, e soluções NoSQL como MongoDB, que é ideal para armazenar grandes volumes de dados não estruturados e aplicações que exigem alta escalabilidade e flexibilidade. Optou-se por usar o PostgreSQL no desenvolvimento do projeto devido à sua capacidade de lidar com transações complexas, suporte a dados estruturados e não estruturados, e forte conformidade com os padrões SQL, o que garante integridade e consistência dos dados.

O PostgreSQL se destaca como uma solução de banco de dados robusta e versátil, adequada para uma ampla gama de aplicações. Sua evolução ao longo dos anos e a capacidade de suportar funcionalidades avançadas o tornam uma escolha excelente para projetos que exigem confiabilidade e flexibilidade. A decisão de utilizá-lo no projeto foi fundamentada em sua capacidade de atender às necessidades específicas de gerenciamento de dados de maneira eficiente e segura.

5.1.4 Typescript

O JavaScript, criado em 1995 por Brendan Eich da Netscape, rapidamente se tornou a linguagem padrão para desenvolvimento web, permitindo a criação de páginas dinâmicas e interativas. No entanto, à medida que aplicações web se tornavam mais complexas, as limitações de JavaScript em termos de tipagem estática e suporte para grandes projetos se tornaram evidentes. Para reduzir essas limitações, a Microsoft lançou o TypeScript em 2012, uma linguagem de programação de código aberto que é um superconjunto (superset) estrito de JavaScript. TypeScript adiciona tipagem estática e outros recursos avançados ao JavaScript, ajudando os desenvolvedores a escrever código mais seguro e escalável.

As principais funções do TypeScript incluem a adição de tipos estáticos, interfaces, classes e módulos, que não existem nativamente no JavaScript. Essas funcionalidades permitem a detecção de erros durante o desenvolvimento, antes do tempo de execução, melhorando a qualidade do código. Além disso, TypeScript se transpila (é convertido) para JavaScript, garantindo compatibilidade total com os navegadores e plataformas que suportam JavaScript. Atualmente, TypeScript é amplamente adotado em projetos de grande escala devido à sua capacidade de melhorar a produtividade e a manutenção do código. Empresas como Google, Microsoft e Airbnb utilizam TypeScript em seus projetos.

No desenvolvimento do projeto backend, optou-se por TypeScript em vez de JavaScript por várias razões. TypeScript proporciona uma melhor experiência de desenvolvimento, oferecendo autocompletar, navegação de código e verificação de tipos em tempo de compilação, o que ajuda a evitar muitos erros comuns em JavaScript. Isso é especialmente importante em projetos backend, onde a robustez e a previsibilidade do código são cruciais. Além disso, TypeScript facilita a colaboração entre desenvolvedores, permitindo que eles entendam e mantenham o código com mais facilidade.

O TypeScript se destaca como uma ferramenta poderosa que complementa e aprimora JavaScript, tornando o desenvolvimento de software mais eficiente e menos propenso a erros. Sua adoção no desenvolvimento da aplicação backend de gerenciamento de reservas e controle automatizado de dispositivos IoT permitiu criar um código mais robusto e sustentável.

5.1.5 Node.js

O Node.js é uma plataforma de desenvolvimento open-source baseada no motor JavaScript V8 do Google Chrome, criada em 2009 por Ryan Dahl. Seu objetivo inicial era possibilitar a execução de JavaScript no lado do servidor, permitindo o desenvolvimento de aplicações web com maior eficiência e escalabilidade. Desde seu

lançamento, o Node.js rapidamente ganhou popularidade devido à sua arquitetura orientada a eventos e sua capacidade de lidar com operações de entrada e saída de forma não bloqueante, o que é particularmente útil para aplicações em tempo real.

Atualmente, o Node.js é amplamente utilizado para construir desde APIs e microservices até aplicações completas de grande escala. Sua capacidade de executar JavaScript tanto no cliente quanto no servidor facilita o desenvolvimento fullstack, enquanto a vasta biblioteca de pacotes disponíveis através do npm (Node Package Manager) oferece soluções para praticamente qualquer necessidade de desenvolvimento. O Node.js tem sido uma escolha popular em diversas indústrias devido à sua performance, escalabilidade e ao suporte contínuo de uma grande comunidade de desenvolvedores.

Uma das principais vantagens do Node.js em comparação com outras soluções é sua natureza assíncrona e orientada a eventos, que permite o gerenciamento eficiente de múltiplas conexões simultâneas sem sobrecarregar os recursos do servidor. Além disso, a utilização de JavaScript, uma linguagem amplamente conhecida e utilizada, reduz a curva de aprendizado para novos desenvolvedores e facilita a integração entre equipes de frontend e backend. A modularidade do Node.js e sua grande comunidade de apoio também contribuem para o desenvolvimento mais rápido e eficiente de aplicações.

Portanto o Node.js se destaca como uma solução versátil e eficiente para o desenvolvimento de aplicações modernas. Sua combinação de performance, escalabilidade e uma vasta gama de ferramentas e bibliotecas tornam-no uma escolha robusta para desenvolvedores que buscam criar aplicações rápidas e escaláveis, atendendo às demandas de um mercado cada vez mais dinâmico e competitivo.

5.1.6 Nest.js

O Nest.js é um framework de desenvolvimento backend criado em 2017 por Kamil Myśliwiec. Inspirado nos princípios de programação modular e fortemente influenciado pelo Angular, o Nest.js foi projetado para oferecer uma estrutura robusta e escalável para a construção de aplicações do lado do servidor. Desde o seu lançamento, o framework tem crescido em popularidade, especialmente entre desenvolvedores que buscam uma abordagem moderna para o desenvolvimento de aplicações backend em Node.js.

O Nest.js fornece uma arquitetura modular que facilita a criação e a manutenção de aplicações escaláveis e bem estruturadas. Ele suporta diversos paradigmas de programação, incluindo orientação a objetos, programação funcional e reativa. Com suporte nativo para TypeScript, o Nest.js oferece tipagem estática, o que melhora a confiabilidade e a legibilidade do código. Atualmente, é amplamente utilizado para construir APIs RESTful, microservices e aplicativos monolíticos, sendo uma escolha

frequente em projetos que exigem alta escalabilidade e flexibilidade.

Entre as principais vantagens do Nest.js estão sua modularidade, que permite a criação de aplicações altamente organizadas e de fácil manutenção, e seu suporte integral a TypeScript, que traz maior segurança e produtividade no desenvolvimento. Em comparação com outros frameworks, como Express.js, o Nest.js oferece uma estrutura mais robusta e orientada a boas práticas, facilitando o desenvolvimento de aplicações complexas. No contexto do projeto de fim de curso, o Nest.js foi escolhido por sua capacidade de suportar a criação de uma aplicação multi-tenant complexa, com necessidades específicas de escalabilidade, organização e integração com outras tecnologias.

O Nest.js se destaca como uma ferramenta poderosa para o desenvolvimento de aplicações backend modernas, combinando a performance do Node.js com uma arquitetura flexível e escalável. Sua adoção neste projeto de fim de curso reflete a busca por soluções eficientes e de alta qualidade, alinhadas com as demandas atuais do mercado de tecnologia.

5.1.7 Docker

O Docker foi lançado em 2013 por Solomon Hykes, inicialmente como um projeto interno da empresa dotCloud. A plataforma foi criada com o objetivo de simplificar o processo de desenvolvimento, distribuição e execução de aplicações por meio da virtualização de contêineres. Desde seu lançamento, o Docker tem revolucionado a maneira como aplicações são desenvolvidas e implantadas, tornando-se uma das ferramentas mais populares no ecossistema de DevOps.

Atualmente, o Docker é amplamente utilizado para criar, distribuir e executar aplicações em contêineres, que são ambientes leves e portáteis que contêm tudo o que uma aplicação precisa para ser executada. Isso inclui código, bibliotecas e dependências, garantindo que a aplicação funcione de maneira consistente em qualquer ambiente. O Docker é uma escolha comum em projetos que exigem portabilidade, escalabilidade e uma integração contínua eficiente, sendo adotado por empresas de todos os tamanhos para modernizar seus fluxos de trabalho de desenvolvimento e implantação.

Entre as principais vantagens do Docker estão a portabilidade de aplicações, a eficiência no uso de recursos e a facilidade de integração com pipelines de CI/CD (integração contínua e entrega contínua). Comparado com máquinas virtuais tradicionais, os contêineres do Docker são mais leves e rápidos, o que resulta em tempos de inicialização menores e melhor utilização de recursos. No contexto deste projeto, o Docker foi escolhido para garantir que o ambiente de desenvolvimento e produção seja consistente, facilitando a implantação em servidores AWS e reduzindo problemas de compatibilidade.

O Docker se destaca como uma solução essencial para o desenvolvimento e a implantação de aplicações modernas, oferecendo uma maneira eficiente de gerenciar ambientes de software. Sua escolha neste projeto reflete a busca por uma solução que ofereça portabilidade, eficiência e flexibilidade, alinhando-se às melhores práticas do mercado e garantindo um processo de desenvolvimento mais ágil e confiável.

5.1.8 Postman

O Postman foi lançado em 2012 por Abhinav Asthana como uma ferramenta auxiliar para o desenvolvimento de APIs. Inicialmente criado como uma extensão para o Google Chrome, o Postman rapidamente evoluiu para uma aplicação completa e independente, tornando-se uma das ferramentas mais populares para o teste e desenvolvimento de APIs. Ao longo dos anos, a plataforma expandiu suas funcionalidades para atender às crescentes demandas de desenvolvedores e equipes de API.

O Postman é amplamente utilizado para testar, documentar e monitorar APIs, oferecendo uma interface amigável que facilita a realização de requisições HTTP, o gerenciamento de coleções de APIs e a automatização de testes. Além disso, ele permite a colaboração em equipe, possibilitando o compartilhamento de coleções e ambientes de teste. Atualmente, o Postman é uma ferramenta indispensável no fluxo de trabalho de desenvolvedores e engenheiros de qualidade, sendo utilizado por milhões de usuários ao redor do mundo.

As vantagens do Postman incluem sua interface intuitiva, que reduz a complexidade do teste de APIs, e suas capacidades avançadas de automação e documentação. Comparado a outras ferramentas, como cURL ou alternativas mais básicas, o Postman oferece uma experiência mais integrada e acessível, especialmente para equipes que precisam colaborar em projetos complexos. No desenvolvimento deste projeto, o Postman foi escolhido por sua facilidade de uso e suas capacidades de automação de testes, o que contribui para um processo de desenvolvimento mais eficiente e menos propenso a erros.

Portanto o Postman se consolida como uma ferramenta essencial para o desenvolvimento e manutenção de APIs, oferecendo funcionalidades abrangentes que facilitam o trabalho de desenvolvedores e equipes de qualidade. Sua utilização neste projeto destaca a importância de ferramentas que otimizam o fluxo de trabalho, garantindo maior eficiência e qualidade no desenvolvimento de sistemas modernos.

5.2 DESENVOLVIMENTO

Após apresentadas as ferramentas e tecnologias selecionadas na Seção 5.1, esta seção tratará do desenvolvimento do projeto backend da aplicação.

5.2.1 Configuração Inicial do Projeto

A primeira etapa do projeto se deu pela configuração do ambiente de desenvolvimento. Para isso, foi o inicializado um novo projeto Nest.js com os comandos

```
nest new sports
```

e instaladas as dependências necessárias. Algumas das bibliotecas usadas estão listadas na Tabela 2.

Tabela 2 – Bibliotecas utilizadas.

Nome	**versão**
@nestjs/cache-manager	2.0.1
@nestjs/common	10.0.0
@nestjs/jwt	10.1.0
@nestjs/platform-express	10.0.0
@nestjs/schedule	4.1.2
@nestjs/typeorm	10.0.0
bcrypt	5.1.1
class-transformer	0.5.1
class-validator	0.14.0
colors	1.4.0
dotenv	16.0.3
ewelink-api-next	1.0.4
moment	2.30.1
pg	8.11.3
reflect-metadata	0.1.13
rxjs	7.8.1
typeorm	0.3.17
uuid	9.0.1

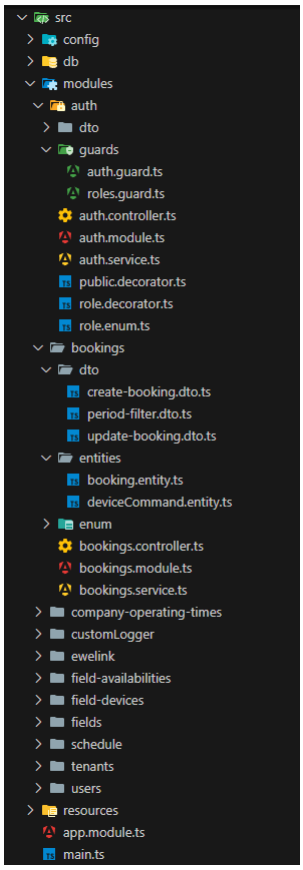
Fonte: Autor.

Em seguida os cointêneres Docker foram configurados para executar o banco de dados PostgreSQL e o PgAdmin (Interface Gráfica para gerenciar o banco de dados PostgreSQL) através de um arquivo docker-compose. Com o ambiente de desenvolvimento configurado, foi realizada a configuração da conexão da aplicação com o banco de dados PostgreSQL com auxílio da biblioteca TypeOrm. O TypeORM é uma biblioteca ORM (Object-Relational Mapping) para TypeScript e JavaScript, que facilita a interação com bancos de dados relacionais. Ele permite a abstração do banco de dados, permitindo que os desenvolvedores se concentrem na manipulação dos dados em vez da implementação detalhada das operações SQL.

5.2.2 Arquitetura de Pastas

A arquitetura do projeto foi dividida em pastas para organizar o código fonte e facilitar a manutenção, seguindo a metodologia de arquitetura modular e de camadas. A pasta "src"contém todos os arquivos do projeto, divididos em pastas para cada funcionalidade ou recurso do sistema. As principais pastas são "config"(configuração do TypeORM com PostgreSQL) , "db"(configuração da conexão do banco PostgreSQL), "modules"(módulos dos recursos da aplicação, como *tenants*, *users* e etc.) e "resources"(como funções de manipulação de dados, *pipes* para criptografia de senhas, *interceptors* para logs de informações e *filters* para tratamento de erros).

Figura 2 – Arquitetura de pastas da aplicação backend.



Fonte: Autor.

5.2.3 Implementação das Funcionalidades

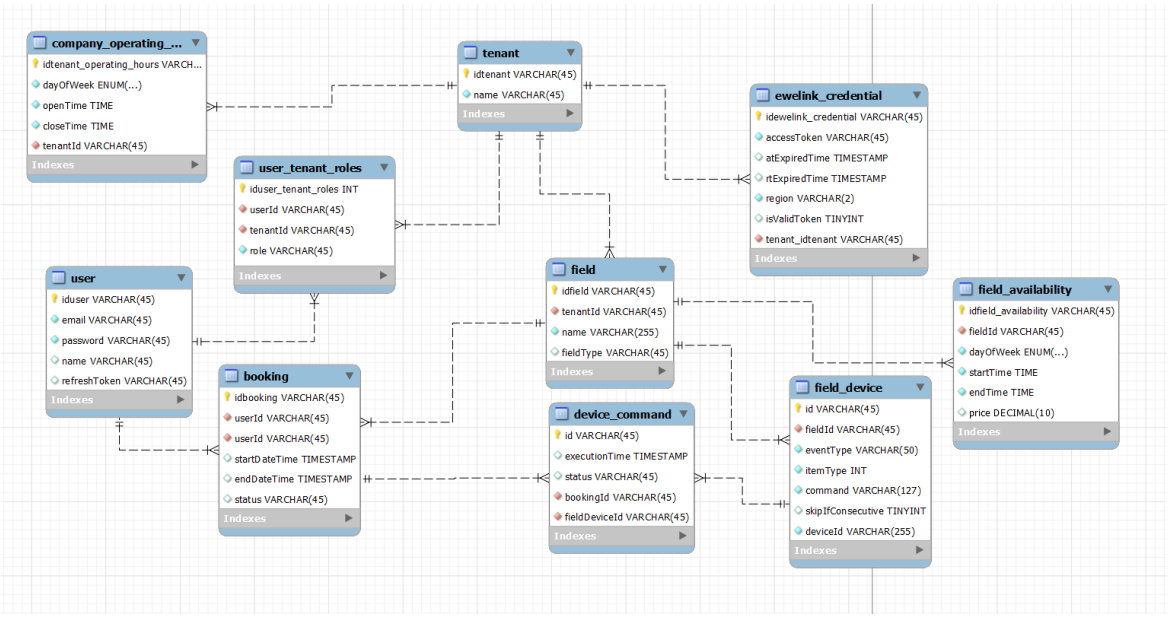
Nessa seção serão detalhadas as implementações das funcionalidades do sistema. Serão abordados o modelo de dados em 5.2.3.1, autenticação e autorização em 5.2.3.2, *tenants* em 5.2.3.3, módulos de usuários em 5.2.3.4, quadras e disponibilidade de quadras em 5.2.3.5, integração com a API eWeLink em 5.2.3.6, quadras-dispositivos

em 5.2.3.7, reservas e comandos IoT em 5.2.3.8 e o envio de comandos aos dispositivos IoT em 5.2.3.9. Algumas rotas e implementações serão mostradas e detalhadas como exemplo e outras ocultadas a fim de reduzir a extensão do texto.

5.2.3.1 Modelagem do Banco de Dados

A primeira etapa do desenvolvimento do sistema é a modelagem do banco de dados. Nesta etapa, são criados os diagramas entidade-relacionamento (ER) que representam as entidades e seus relacionamentos. O diagrama ER é uma representação visual das tabelas do banco de dados e dos relacionamentos entre elas e está apresentado na Figura 3.

Figura 3 – Modelagem do Banco de Dados.



Fonte: Autor.

tenant (Locatário/Empresa)

Armazena informações das empresas que utilizam o sistema. Cada empresa tem um ambiente isolado para suas configurações e dados.
Relacionamentos:

- Relaciona-se com `field` (uma empresa pode ter várias quadras).
- Relaciona-se com `user_tenant_roles` (definição de papéis dos usuários dentro da empresa).
- Relaciona-se com `ewelink_credential` (armazenamento de credenciais IoT por empresa).

- Relaciona-se com `company_operating_hours` (definição do horário de funcionamento da empresa).

user (Usuário do Sistema)

Cadastra usuários na plataforma, permitindo o acesso e gestão de informações dos usuários em diferentes locatários.
Relacionamentos:

- Relaciona-se com `use_tenant_roles` (um usuário pode ter diferentes papéis em empresas distintas).
- Relaciona-se com `booking` (usuários realizam reservas de quadras).

user_tenant_roles (Papéis de Usuário na Empresa)

Define os papéis dos usuários em diferentes locatários, garantindo acesso e permissões personalizadas para cada perfil.
Relacionamentos:

- Relaciona-se com `user` (indica o usuário).
- Relaciona-se com `tenant` (indica a empresa onde o usuário tem um papel).

company_operating_hours (Horários de Funcionamento da Empresa)

Define os horários de funcionamento das empresas, permitindo customização e gestão do tempo disponível para operações em cada locatário.
Relacionamentos:

- Relaciona-se com `tenant` (cada locatário tem seus próprios horários de funcionamento).

field (Quadras Esportivas)

Armazena as quadras disponíveis para reservas, relacionadas ao locatário que as possui e contendo informações específicas sobre cada quadra.
Relacionamentos:

- Relaciona-se com `tenant` (cada locatário pode ter várias quadras).
- Relaciona-se com `field_availability` (definição de horários disponíveis para a quadra).
- Relaciona-se com `field_device` (dispositivos IoT associados à quadra).
- Relaciona-se com `booking` (quadras podem ser reservadas).

field_availability (Disponibilidade da Quadra)

Determina os horários disponíveis para cada quadra, indicando o período de tempo em que a mesma fica aberta e quanto custa o uso dela.
Relacionamentos:

- Relaciona-se com `field` (define a disponibilidade para cada quadra).

booking (Reservas de Quadras)

Registra as reservas feitas pelos usuários, associando-as ao locatário que possui a quadra reservada e outros detalhes sobre o horário, usuário e status da reserva.
Relacionamentos:

- Relaciona-se com `user` (quem fez a reserva).
- Relaciona-se com `field` (qual quadra foi reservada).

ewelink_credential (Credenciais para Controle IoT via eWeLink)

Armazena informações de autenticação da integração entre o sistema e dispositivos IoT, como tokens de acesso e gerenciamento de recursos.
Relacionamentos:

- Relaciona-se com `tenant` (cada locatário pode ter credenciais IoT).

field_device (Dispositivos IoT Associados às Quadras)

Registra os dispositivos IoT instalados em cada quadra, indicando seu tipo de funcionalidade, comando a ser executado e outros detalhes relevantes para o gerenciamento.
Relacionamentos:

- Relaciona-se com `field` (cada quadra pode ter vários dispositivos IoT).

device_command (Comandos Enviados a Dispositivos IoT)

Armazena os comandos programados ou executados para dispositivos IoT, associados às reservas e que devem ser executados conforme a agenda de reservas.
Relacionamentos:

- Relaciona-se com `booking` (comandos são acionados conforme reservas).
- Relaciona-se com `field_device` (cada comando pertence a um dispositivo IoT).

5.2.3.2 Autenticação e Autorização

Visão Geral

O módulo de autenticação é responsável por garantir a segurança e o controle de acesso ao sistema multi-tenant. Ele permite que os usuários se autentiquem utilizando

credenciais seguras e fornece mecanismos para gerenciamento de tokens de acesso e renovação de sessão. O sistema utiliza autenticação baseada em tokens JWT (*JSON Web Token*) e cookies para armazenar *refresh tokens* com segurança.

A autenticação dos usuários é fundamental para garantir que apenas indivíduos autorizados tenham acesso aos recursos do sistema. Além disso, o módulo gerencia o login, o registro de novos usuários (inicialmente sem vínculo com nenhum *tenant*), a renovação de tokens e o logout seguro, invalidando tokens de sessão comprometidos.

Fluxo das Rotas

O controlador de autenticação (`auth.controller.ts`) define os endpoints responsáveis pelo login, registro de novos usuários, renovação de tokens e logout. Ele interage com o serviço de autenticação (`auth.service.ts`), que contém a lógica de geração e validação de tokens, verificação de credenciais e manipulação segura de sessões.

5.2.3.2.1 Autenticação de Usuário

- **Rota:** `POST /auth/login`
- **Descrição:** Permite que um usuário se autentique utilizando seu e-mail e senha. Retorna um *access token* e um *refresh token*.
- **Fluxo:**
  1. O controlador recebe as credenciais do usuário.
  2. O serviço de autenticação busca o usuário no banco de dados e valida a senha.
  3. Se as credenciais forem válidas, um novo *access token* e um *refresh token* são gerados e retornados.

O código construído para o *controller* e o *service* estão disponíveis no Código 5.1 e no Código 5.2, respectivamente, e servirão de exemplo para os demais *controllers* e *services*.

```
1  @Post('login')
2  async login(
3    @Res({ passthrough: true }) res: Response,
4    @Body() { email, password }: AuthenticateDto
5  ) {
6    const { loggedUser, accessToken, refreshToken } =
7      await this.authService.login(email, password)
8
9    res.cookie('jwt', refreshToken, {
10     httpOnly: true,
11     sameSite: 'none',
12     secure: true,
```

```
      maxAge: 24 * 60 * 60 * 1000
    })
    res.status(201)
    return { accessToken, ...loggedUser }
  }
```

Código 5.1 – Exemplo de *controller* para *login*.

```
async login(email: string, insertedPassword: string) {
  try {
    const foundUser = await this.userService.searchForEmail(email)

    const userWasAuthenticated = await bcrypt.compare(
      insertedPassword,
      foundUser.password
    )

    if (!userWasAuthenticated)
      throw new UnauthorizedException('E-mail e/ou senha incorretos.')

    const payload: UserPayload = {
      sub: foundUser.id,
      email: foundUser.email,
      name: foundUser.name
    }

    const newRefreshToken = await this.createRefreshToken(payload)
    const newAccessToken = await this.createAccessToken(payload)

    await this.updateRefreshToken(newRefreshToken, {
      sub: foundUser.id,
      ...foundUser
    })

    return {
      accessToken: newAccessToken,
      refreshToken: newRefreshToken,
      loggedUser: payload
    }
  } catch (e) {
    if (e?.status === 404 || e?.status === 401)
      throw new UnauthorizedException('E-mail e/ou senha incorretos.')

    console.log('Erro interno:', e)
    throw new InternalServerErrorException(
      'Erro interno no servidor. ' + e?.message
    )
  }
```

```
41  }
```

Código 5.2 – Exemplo de *service* para *login*.

5.2.3.2.2 Registro de Usuário

- **Rota:** `POST /auth/register`
- **Descrição:** Permite criar um novo usuário no sistema sem vinculação com nenhuma empresa, armazenando suas credenciais de forma segura.
- **Fluxo:**
  1. O controlador recebe os dados do novo usuário.
  2. A senha é criptografada antes de ser salva no banco de dados.
  3. O usuário é criado e retornado na resposta.

5.2.3.2.3 Renovação de Token

- **Rota:** `GET /auth/refresh`
- **Descrição:** Permite gerar um novo *access token* utilizando um *refresh token* válido armazenado em um cookie.
- **Fluxo:**
  1. O controlador verifica a presença do *refresh token* nos cookies da requisição.
  2. O serviço valida o token e gera um novo *access token*.
  3. O novo token é retornado na resposta.

5.2.3.2.4 Logout

- **Rota:** `GET /auth/logout`
- **Descrição:** Invalida a sessão do usuário removendo seu *refresh token* armazenado.
- **Fluxo:**
  1. O controlador verifica o *refresh token* armazenado no cookie da requisição.
  2. O serviço de autenticação invalida o token no banco de dados.
  3. O cookie é removido da resposta e a sessão é encerrada.

5.2.3.3 Módulo de Tenants

Visão Geral

O módulo de *tenants* é responsável por gerenciar múltiplos locatários dentro do sistema, permitindo a criação, consulta, atualização e remoção de *tenants*. Esse módulo é essencial para suportar um ambiente multi-tenant, onde cada empresa ou cliente pode operar de forma isolada dentro da mesma aplicação.

As principais funcionalidades deste módulo incluem:

- Criação de um *tenant*, associando-o a um usuário administrador.
- Consulta de *tenants*, tanto de forma individual quanto em lista.
- Atualização das informações de um *tenant*.
- Remoção de um *tenant*.
- Controle de permissões para diferentes níveis de usuário dentro de um *tenant*.

O acesso às rotas do módulo é protegido por um sistema de autenticação e autorização baseado em *roles*, garantindo que apenas usuários autorizados possam executar determinadas ações.

Fluxo das Rotas

As rotas do módulo de *tenants* são definidas no `TenantsController`, garantindo a estruturação adequada dos endpoints da API. A seguir, estão dispostos os fluxos de duas das principais operações: criação e busca de *tenants*.

5.2.3.3.1 Criação de um Tenant

- **Rota:** `POST /tenants`
- **Descrição:** Criação de um novo *tenant*.
- **Fluxo:**
  1. O usuário autenticado envia uma requisição contendo os dados do novo *tenant*.
  2. O serviço verifica se o usuário tem permissão para criar um *tenant*.
  3. Se autorizado, os dados do *tenant* são persistidos no banco de dados.
  4. O usuário criador recebe automaticamente o papel de administrador dentro do novo *tenant*.
  5. A resposta retorna uma mensagem de sucesso e os detalhes do *tenant* criado.

5.2.3.3.2 Consulta de um Tenant

- **Rota:** `GET /tenants/:id`
- **Descrição:** Buscar um *tenant* específico.
- **Fluxo:**
  1. O usuário faz uma requisição com o identificador do *tenant* desejado.
  2. O serviço consulta o banco de dados para localizar o *tenant* correspondente.
  3. Se encontrado, os detalhes do *tenant* são retornados na resposta.
  4. Se o *tenant* não for encontrado, uma exceção é lançada informando o erro.

Outras Operações

Além das rotas descritas acima, o módulo também disponibiliza endpoints para listar todos os *tenants*, atualizar informações e remover um *tenant*. Estas operações são protegidas por um sistema de autenticação e autorização baseado em *roles*, garantindo que apenas usuários com as permissões adequadas possam realizá-las.

5.2.3.4 Módulo de Usuarios

Visão Geral

O módulo de usuários é responsável por gerenciar os perfis de usuários dentro do sistema multi-tenant. Ele permite o cadastro, consulta, edição e remoção de usuários (vinculados a uma empresa específica), além de possibilitar a gestão de papéis dentro de diferentes empresas. O sistema suporta diferentes níveis de acesso e autenticação baseada em tokens.

Os usuários podem pertencer a múltiplas empresas e ter diferentes papéis em cada uma delas. Isso permite um controle granular das permissões e acesso às funcionalidades do sistema.

Fluxo das Rotas

O controlador de usuários (`users.controller.ts`) define os endpoints para interação com o serviço de usuários (`users.service.ts`), que contém a lógica de negócio. O código construído para o *controller* e o *service* estão disponíveis no Código 5.3 e no Código 5.4, respectivamente, e servirão de exemplo para os demais *controllers* e *services*.

5.2.3.4.1 Criação de Usuário

- **Rota:** `POST /tenant/:tenantId/users`

- **Descrição:** Permite criar um novo usuário, recebendo os dados básicos no corpo da requisição. Caso seja informado um `tenantId`, o usuário é vinculado a essa empresa.
- **Fluxo:**
  1. O controlador recebe os dados do novo usuário.
  2. O serviço cria a entidade e a salva no banco.
  3. O usuário é retornado na resposta.

```
@Post()
async createUser(
  @Body() { password, ...userData }: CreateUserDTO,
  @Body('password', HashingPasswordPipe) hashedPassword: string
) {
  const createdUser = await this.userService.createUser({
    ...userData,
    password: hashedPassword
  })

  return {
    message: 'Usuario criado!',
    user: new ListUserDTO({ id: createdUser.id, name: createdUser.name
        })
  }
}
```

Código 5.3 – Exemplo de *controller* para criação de usuário.

```
async createUser(userData: CreateUserDTO, tenantId?: string) {
  const userEntity = new User()

  Object.assign(userEntity, userData as User)
  const createdUser = await this.userRepository.save(userEntity)

  console.log('Create relation to tenant: ', tenantId)

  return createdUser
}
```

Código 5.4 – Exemplo de *service* para criação de usuário.

5.2.3.4.2 Consulta de Usuário por ID

- **Rota:** `GET /tenant/:tenantId/users/:id`

- **Descrição:** Retorna as informações detalhadas de um usuário, incluindo seus papéis nas empresas.
- **Fluxo:**
  1. O controlador recebe o `id` do usuário via parâmetro de rota.
  2. O serviço busca o usuário no banco, carregando as relações necessárias.
  3. Caso não seja encontrado, retorna erro.
  4. Se encontrado, retorna os dados do usuário.

5.2.3.4.3 Listagem de Usuários de uma Empresa

- **Rota:** `GET /tenant/:tenantId/users`
- **Descrição:** Retorna uma lista de usuários vinculados a uma empresa.
- **Fluxo:**
  1. O controlador recebe o `tenantId` como um parâmetro.
  2. O serviço busca os usuários que possuem relação com essa empresa.
  3. Retorna a lista formatada.

5.2.3.4.4 Atualização de Usuário

- **Rota:** `PATCH /tenant/:tenantId/users/:id`
- **Descrição:** Permite a edição dos dados cadastrais de um usuário.
- **Fluxo:**
  1. O controlador recebe o `id` e os novos dados do usuário.
  2. O serviço verifica se o usuário existe.
  3. Caso não exista, retorna erro.
  4. Atualiza os dados e os salva no banco.
  5. Retorna os novos dados do usuário.

5.2.3.4.5 Atualização de Papel de Usuário

- **Rota:** `PATCH /tenant/:tenantId/users/:id/role`
- **Descrição:** Atualiza o papel do usuário dentro de uma empresa.
- **Fluxo:**

1. O controlador recebe o `id` do usuário, o `tenantId` e o novo papel.
2. O serviço verifica se o papel existe.
3. O serviço verifica se o usuário solicitante tem permissão para a alteração.
4. Caso não tenha permissão, retorna erro.
5. Atualiza o papel e salva no banco.
6. Retorna os novos dados do usuário.

5.2.3.4.6 Exclusão de Usuário

- **Rota:** `DELETE /tenant/:tenantId/users/:id`
- **Descrição:** Remove um usuário do sistema (exclusão lógica).
- **Fluxo:**

1. O controlador recebe o `id` do usuário.
2. O serviço verifica se o usuário existe.
3. Caso não exista, retorna erro.
4. Apaga logicamente o registro do banco.
5. Retorna sucesso.

Conclusão

Esse módulo é essencial para a gestão dos usuários do sistema, garantindo que cada um tenha acesso adequado aos recursos da empresa em que está vinculado. As regras de permissão asseguram que apenas usuários autorizados possam modificar informações sensíveis, proporcionando segurança e integridade aos dados.

5.2.3.5 Módulos de Quadras e Disponibilidade de Quadras

Visão Geral

Os módulos de *quadras* e *disponibilidade de quadras* são responsáveis pelo gerenciamento dos espaços esportivos disponíveis para reserva dentro do sistema. O módulo de quadras gerencia a criação, consulta, atualização e remoção das quadras cadastradas, enquanto o módulo de disponibilidade de quadras permite definir os horários disponíveis para cada quadra em dias específicos da semana.

As principais funcionalidades destes módulos incluem:

- Criação e gerenciamento das quadras dentro de um *tenant*.
- Definição de horários disponíveis para cada quadra.

- Consulta das quadras e suas disponibilidades.
- Atualização e remoção de quadras e horários disponíveis.

Fluxo das Rotas

As rotas desses módulos são definidas nos controladores `FieldsController` e `FieldAvailabilit`
A seguir, estão dispostos os fluxos das operações de criação e consulta de quadras e disponibilidade.

5.2.3.5.1 Criação de uma Quadra

- **Rota:** `POST /tenants/:tenantId/fields`
- **Descrição:** Criar uma nova quadra dentro de um *tenant*.
- **Fluxo:**
  1. O usuário autenticado com permissão faz uma requisição contendo os dados da nova quadra.
  2. O serviço verifica se o *tenant* informado existe.
  3. Se autorizado, os dados da quadra são persistidos no banco de dados.
  4. A resposta retorna uma mensagem de sucesso e os detalhes da quadra criada.

5.2.3.5.2 Consulta de uma Quadra

- **Rota:** `GET /tenants/:tenantId/fields/:id`
- **Descrição:** Buscar uma quadra específica dentro de um *tenant*.
- **Fluxo:**
  1. O usuário faz uma requisição com o identificador da quadra e do *tenant*.
  2. O serviço consulta o banco de dados para localizar a quadra correspondente.
  3. Se encontrada, os detalhes da quadra são retornados na resposta.
  4. Se a quadra não for encontrada, uma exceção é lançada informando o erro.

5.2.3.5.3 Criação de Disponibilidade de Quadra

- **Rota:** `POST /tenants/:tenantId/fields/:fieldId/field-availabilities`
- **Descrição:** Criar um ou mais horários disponíveis para uma quadra específica.
- **Fluxo:**
  1. O usuário autenticado com permissão faz uma requisição contendo os horários disponíveis para uma quadra.
  2. O serviço verifica se a quadra informada existe.
  3. Se validado, os horários são armazenados no banco de dados.
  4. A resposta retorna a confirmação da criação dos horários.

5.2.3.5.4 Consulta de Disponibilidade de Quadra

- **Rota:** `GET /tenants/:tenantId/fields/:fieldId/field-availabilities/:id`
- **Descrição:** Buscar um horário de disponibilidade específico de uma quadra.
- **Fluxo:**
  1. O usuário faz uma requisição com o identificador da disponibilidade e da quadra correspondente.
  2. O serviço consulta o banco de dados para localizar o horário correspondente.
  3. Se encontrado, os detalhes da disponibilidade são retornados na resposta.
  4. Se o horário não for encontrado, uma exceção é lançada informando o erro.

Outras Operações

Além das rotas descritas acima, os módulos também disponibilizam endpoints para listar todas as quadras, atualizar informações e remover quadras e horários disponíveis.

5.2.3.6 Integração com eWeLink

Visão Geral

O módulo de integração com a API eWeLink é responsável por permitir a conexão entre o sistema e dispositivos IoT suportados pela plataforma eWeLink. Ele possibilita que os administradores das empresas associem sua conta à plataforma, consultem os seus dispositivos e interajam de forma automatizada com os equipamentos cadastrados.

As principais funcionalidades deste módulo incluem:

- Autenticação de usuários na API eWeLink via OAuth2.0 para permitir o gerenciamento dos dispositivos pela plataforma.
- Consulta e listagem dos dispositivos vinculados a um determinado *tenant*.
- Atualização e renovação automática dos tokens de acesso à API eWeLink.
- Controle seguro de permissões, garantindo que apenas administradores possam gerenciar dispositivos.

Fluxo das Rotas

As rotas deste módulo são definidas no `EwelinkController` e interagem com o serviço de autenticação e gerenciamento de dispositivos eWeLink, fornecido pelo `EwelinkService`. A seguir, apresenta-se o fluxo das operações deste módulo.

5.2.3.6.1 Autenticação na API eWeLink

- **Rota:** `GET /tenants/:tenantId/ewelink/login`
- **Descrição:** Redireciona o usuário para a página de autenticação da eWeLink.
- **Fluxo:**
  1. O usuário autenticado faz uma requisição para iniciar a autenticação com a API eWeLink.
  2. O serviço gera uma URL de autenticação com a informação do `tenantId` e redireciona o usuário para a página de login da eWeLink.
  3. Após a autenticação bem-sucedida, a eWeLink redireciona o usuário de volta ao sistema com um código de autorização através do `tenantId`.

Esse fluxo de autenticação segue o formato OAuth2.0. *Open Authorization* (OAuth) 2.0 é um protocolo padrão para autorização de acesso a recursos online. Ele permite que um cliente solicite permissão específica do usuário para o acesso a dados ou serviços, sem compartilhar informações sensitivas diretamente com o provedor de recursos. OAuth2.0 define quatro protocolos:

1. Autorização de Cliente (Client Authorization): o cliente solicita permissão ao usuário para autorizar o acesso à aplicação web ou dispositivo móvel que deseja usar os recursos do provedor de recursos.
2. Tokens de Acesso (Access Tokens): o provedor de recursos concede um token de acesso ao cliente, que é utilizado para autenticar futuras solicitações e obter acesso aos recursos protegidos.

3. Refresh Tokens: quando o token de acesso expira ou está prestes a fazer isso, o cliente pode usar o refresh token para obter um novo token sem reautenticar o usuário.

4. Gerenciamento de Autorização (Authorization Management): o provedor de recursos permite que os administradores revoguem ou gerenciem as permissões concedidas a um cliente.

O fluxo geral do protocolo OAuth2.0 é o seguinte:

1. Autenticação do Usuário: o usuário realiza autenticação no provedor de recursos.

2. Solicitando Autorização: a aplicação web ou dispositivo móvel solicita permissão ao usuário para acessar um recurso protegido em nome do usuário.

3. Autorização de Cliente: o provedor de recursos exibe uma janela à qual o cliente pode dar autorização à aplicação web ou dispositivo móvel.

4. Gerando Tokens de Acesso e Refresh Tokens: se o usuário concede a autorização, o provedor de recursos gera um token de acesso e um refresh token (se for o caso).

5. Recebendo e Utilizando Tokens de Acesso: a aplicação web ou dispositivo móvel envia o token de acesso para solicitar os recursos protegidos.

6. Verificando e Renovando Tokens de Acesso: quando o token de acesso expira, a aplicação web ou dispositivo móvel usa o refresh token para obter um novo token sem reautenticar o usuário.

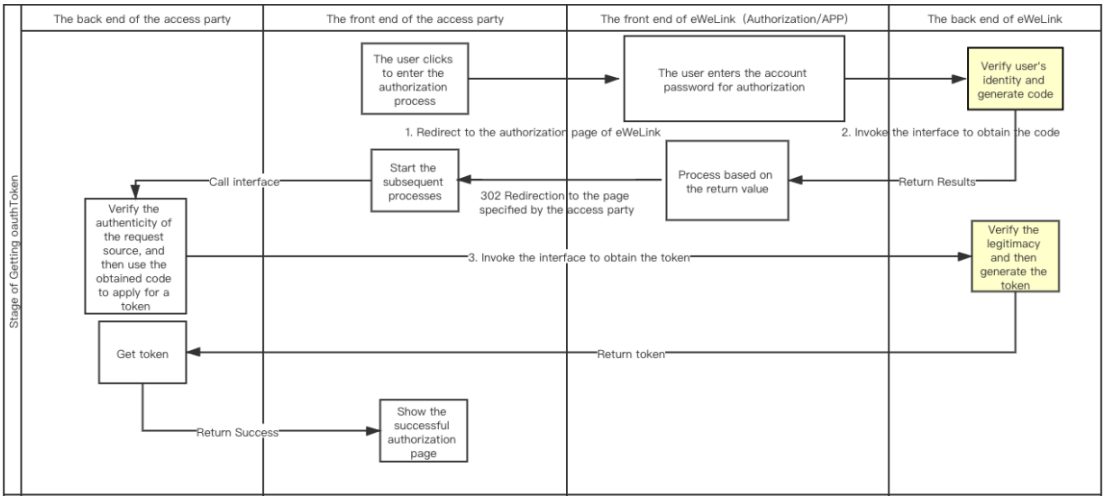
OAuth2.0 é amplamente utilizado em plataformas online, como Google Drive, Facebook e Twitter, que promove a autorização segura de recursos entre diferentes aplicações.

No contexto da aplicação desenvolvida neste PFC, o usuário é redirecionado para a página de autenticação da eWeLink, onde ele pode inserir suas credenciais. Após o login bem-sucedido, a eWeLink redireciona o usuário de volta ao sistema com um código de autorização (*access token*) conforme demonstrado na Figura 4, que é armazenado no banco de dados para permitir interações futuras com a API.

5.2.3.6.2 Obtenção do Token de Acesso eWeLink

- **Rota:** `GET /ewelink/redirectUrl`
- **Descrição:** Recebe o código de autorização da eWeLink e gera um token de acesso.

Figura 4 – OAuth2.0 EWeLink. Obter oauth *token*



Fonte: (Shenzhen Coolkit Technology CO., 2024)

- **Fluxo:**

  1. O usuário é redirecionado da eWeLink para o sistema com um código de autorização e o id do *tenant*.
  2. O sistema utiliza esse código para obter um token de acesso e um token de atualização.
  3. As credenciais são armazenadas no banco de dados usando o id do *tenant* para permitir interações futuras com a API.

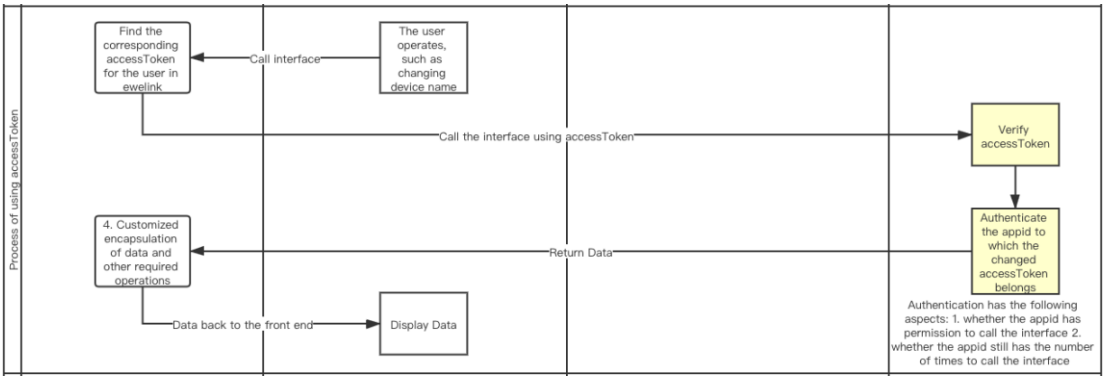
5.2.3.6.3 Listagem de Dispositivos eWeLink

- **Rota:** `POST /tenants/:tenantId/ewelink/listAllThings`
- **Descrição:** Lista todos os dispositivos associados a um determinado *tenant*.
- **Fluxo:**

  1. O usuário autenticado com permissão faz uma requisição para listar os dispositivos.
  2. O serviço verifica as credenciais e atualiza os tokens se necessário.
  3. O serviço consulta a API eWeLink e retorna a lista de dispositivos cadastrados.

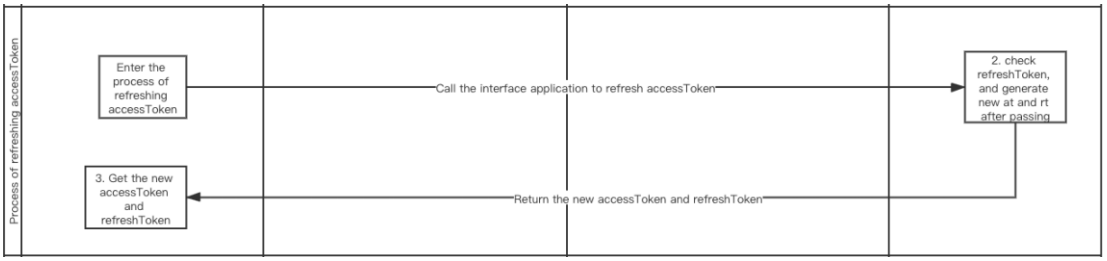
A Figura 5 mostra o processo de uso do *access token* na interação com a plataforma *Coolkit Open Platform 4.3 EWeLink*. Neste exemplo, o usuário autenticado

Figura 5 – OAuth2.0 EWeLink. Usando um *access token*



Fonte: (Shenzhen Coolkit Technology CO., 2024)

Figura 6 – OAuth2.0 EWeLink. Atualizando um *access token*



Fonte: (Shenzhen Coolkit Technology CO., 2024)

com permissão realiza uma requisição para listar os dispositivos, que são consultados na API eWeLink e retornados ao usuário.

Caso o *access token* esteja expirado ou inválido, a aplicação solicita um novo token utilizando o *refresh token*. O servidor de autorização gera um novo *access token* e um novo *refresh token*, que podem ser usados para futuras requisições. Caso o *refresh token* esteja expirado ou inválido, a plataforma acusará erro e indicará ao usuário para efetuar login novamente para obter um novo *access token* e *refresh token*.

Outras Operações

Além das rotas descritas acima, o módulo também inclui funcionalidades para atualização de tokens de acesso, e controle dos dispositivos IoT para fins de testes (os controles da plataforma serão enviados pelo módulo *schedule* demonstrado na Seção 5.2.3.9).

5.2.3.7 Módulo de Quadras-Dispositivos

Visão Geral

O módulo de *field-devices* é responsável pela integração entre as quadras esportivas

e os dispositivos IoT, permitindo o gerenciamento dos equipamentos eletrônicos associados a cada quadra. Essa funcionalidade possibilita o controle automatizado de iluminação, climatização e outros dispositivos conectados, otimizando a experiência do usuário e a gestão operacional do sistema.

As principais funcionalidades deste módulo incluem:

- Associação de comandos de dispositivos a quadras esportivas específicas.
- Consulta e listagem dos comandos de dispositivos cadastrados para uma quadra.
- Atualização e remoção de dispositivos vinculados às quadras.
- Controle seguro de permissões, garantindo que apenas administradores e gerentes possam gerenciar os dispositivos.

Fluxo das Rotas

As rotas deste módulo são definidas no `FieldDevicesController` e interagem com o serviço de gerenciamento de dispositivos `FieldDevicesService`. A seguir, são apresentados os fluxos de todas as operações disponíveis.

5.2.3.7.1 Criação de um Dispositivo de Quadra

- **Rota:** `POST /tenants/:tenantId/fields/:fieldId/field-devices`
- **Descrição:** Criar um novo comando de dispositivo vinculado a uma quadra esportiva.
- **Fluxo:**
  1. O usuário autenticado com permissão faz uma requisição contendo os dados do dispositivo obtidos através da API eWeLink juntamente com o comando desejado.
  2. O serviço verifica se a quadra informada existe.
  3. Se validado, os dados do dispositivo e o comando desejado são armazenados no banco de dados e associados à quadra.
  4. A resposta retorna uma mensagem de sucesso e os detalhes do dispositivo cadastrado.

5.2.3.7.2 Consulta de Todos os Dispositivos de uma Quadra

- **Rota:** `GET /tenants/:tenantId/fields/:fieldId/field-devices`
- **Descrição:** Retorna uma lista com todos os dispositivos vinculados a uma quadra específica.

- **Fluxo:**
    1. O usuário faz uma requisição para listar os dispositivos de uma quadra específica.
    2. O serviço busca todos os dispositivos associados à quadra no banco de dados.
    3. A lista dos dispositivos cadastrados é retornada na resposta.

5.2.3.7.3 Consulta de um Dispositivo de Quadra Específico

- **Rota:** `GET /tenants/:tenantId/fields/:fieldId/field-devices/:id`
- **Descrição:** Buscar um dispositivo específico vinculado a uma quadra esportiva.
- **Fluxo:**
    1. O usuário faz uma requisição com o identificador do dispositivo e da quadra correspondente.
    2. O serviço consulta o banco de dados para localizar o dispositivo correspondente.
    3. Se encontrado, os detalhes do dispositivo são retornados na resposta.
    4. Se o dispositivo não for encontrado, uma exceção é lançada informando o erro.

5.2.3.7.4 Atualização de um Dispositivo de Quadra

- **Rota:** `PATCH /tenants/:tenantId/fields/:fieldId/field-devices/:id`
- **Descrição:** Atualiza as informações de um comando de dispositivo vinculado a uma quadra esportiva.
- **Fluxo:**
    1. O usuário autenticado com permissão faz uma requisição contendo os dados atualizados do dispositivo.
    2. O serviço verifica se o dispositivo informado existe.
    3. Se validado, os dados do dispositivo são atualizados no banco de dados.
    4. A resposta retorna uma mensagem de sucesso e os detalhes do dispositivo atualizado.

5.2.3.7.5 Remoção de um Dispositivo de Quadra

- **Rota:** `DELETE /tenants/:tenantId/fields/:fieldId/field-devices/:id`
- **Descrição:** Remove um comando de dispositivo vinculado a uma quadra esportiva.
- **Fluxo:**
  1. O usuário autenticado com permissão faz uma requisição para remover o dispositivo.
  2. O serviço verifica se o dispositivo informado existe.
  3. Se encontrado, o dispositivo é removido logicamente do banco de dados.
  4. A resposta confirma a remoção do dispositivo.

5.2.3.8 Módulo de Reservas e Comandos IoT

Visão Geral

O módulo de *reservas e comandos IoT* é responsável pelo gerenciamento das reservas de quadras esportivas e pela automação dos dispositivos IoT associados. Ele garante que os usuários possam realizar reservas de maneira eficiente, ao mesmo tempo que controla dispositivos como iluminação e climatização, otimizando os recursos disponíveis.

As principais funcionalidades deste módulo incluem:

- Criação e gerenciamento de reservas de quadras esportivas.
- Listagem de reservas existentes, incluindo filtros por período e usuário.
- Controle automatizado de dispositivos IoT com base nos horários das reservas.
- Prevenção de conflitos de horários e gerenciamento de comandos consecutivos para dispositivos.

Fluxo das Rotas

As rotas deste módulo são definidas no `BookingsController` e interagem com o serviço de gerenciamento de reservas e automação de dispositivos IoT, fornecido pelo `BookingsService`. A seguir, são apresentados os fluxos das principais operações disponíveis.

5.2.3.8.1 Criação de uma Reserva

- **Rota:** `POST /tenants/:tenantId/fields/:fieldId/bookings`
- **Descrição:** Criar uma nova reserva para uma quadra esportiva.
- **Fluxo:**
  1. O usuário autenticado com permissão faz uma requisição contendo os detalhes da reserva.
  2. O serviço verifica se a quadra informada existe e se a disponibilidade está correta.
  3. Caso não haja conflitos de horário, a reserva é criada e armazenada no banco de dados.
  4. Se houver dispositivos IoT associados à quadra, são criados comandos automáticos para acioná-los no início e no final da reserva.
  5. A resposta retorna uma mensagem de sucesso com os detalhes da reserva e dos comandos IoT gerados.

O código construído para o o *service* está disponíveis no Código 5.5.

```
async create(
  createBookingDto: CreateBookingDto,
  fieldId: string,
  userId: string
) {
  const fieldAvailability = await this.fieldAvailabilitiesService.
     findOne(
    fieldId,
    createBookingDto.fieldAvailabilityId
  )

  const user = await this.usersService.findById(userId)

  const startDateTime = moment(createBookingDto.startTime)
  const endDateTime = moment(createBookingDto.endTime)
  console.log('startDateTime moment', startDateTime)
  console.log('endDateTime moment', endDateTime)

  const hasConflict = await this.hasBookingConflict(
    fieldId,
    createBookingDto.startTime,
    createBookingDto.endTime
  )

```

```
24      if (hasConflict)
25        throw new ConflictException('Ja existe uma reserva neste periodo')
26
27      // check if the week days are the same
28      if (
29        startDateTime.utc().day() !==
30        moment().day(fieldAvailability.dayOfWeek).day()
31      )
32        throw new BadRequestException(
33          'Dia da semana solicitado nao confere com o dia disponivel'
34        )
35
36      // check if the start and end times match the availability range
37      if (
38        startDateTime.utc().format('HH:mm:ss') !== fieldAvailability.
             startTime ||
39        endDateTime.utc().format('HH:mm:ss') !== fieldAvailability.endTime
40      ) {
41        console.log('hora disponivel inicio:', fieldAvailability.startTime
             )
42        console.log(
43          'hora solicitada inicio:',
44          startDateTime.utc().format('HH:mm:ss')
45        )
46        console.log('hora disponivel final:', fieldAvailability.endTime)
47        console.log(
48          'hora solicitada final:',
49          endDateTime.utc().format('HH:mm:ss')
50        )
51
52        throw new BadRequestException(
53          'Horarios solicitados nao conferem com os horarios disponiveis'
54        )
55      }
56
57      const booking = new Booking()
58
59      Object.assign(booking, {
60        ...createBookingDto,
61        field: fieldAvailability.field,
62        user
63      } as unknown as Booking)
64      // Filter data to return without user for the booking
65      const createdBooking = await this.bookingsRepository.save(booking)
66
67      const fieldDevicesByField = await this.fieldDevicesService.findAll(
           fieldId)
```

```
68
69      // if there are field devices, create device commands
70      const returnObject = { booking: createdBooking }
71      if (fieldDevicesByField.length > 0) {
72        const bookingDeviceCommands = fieldDevicesByField.map((device) =>
              {
73          const deviceCommand = new DeviceCommand()
74          Object.assign(deviceCommand, {
75            booking: createdBooking,
76            fieldDevice: device,
77            executionTime:
78              device.eventType === EventTypeEnum.START
79                ? booking.startTime
80                : booking.endTime
81          })
82          return deviceCommand
83        })
84
85        const consecutiveCommands = await Promise.all(
86          bookingDeviceCommands.map((command) => {
87            // Step 1: Check if there are any consecutive commands for the
                  same device and ignore them
88            const query = {
89              where: {
90                status: CommandStatusEnum.PENDING,
91                fieldDevice: {
92                  deviceId: command.fieldDevice.deviceId,
93                  skipIfConsecutive: true,
94                  field: { id: fieldId }
95                },
96                executionTime: Between(
97                  moment(command.executionTime).subtract(5, 'minutes').
                       toDate(),
98                  moment(command.executionTime).add(5, 'minutes').toDate()
99                )
100             },
101             relations: { fieldDevice: { field: true } }
102           }
103           return this.deviceCommandsRepository.findOne(query)
104         })
105       )
106
107       // search if the command has any consecutive command so it needs
            to be ignored
108       const deviceCommandsWithoutConsecutiveCommands =
109         bookingDeviceCommands.filter(
110           (_command, index) => !consecutiveCommands[index]
```

```
111         )
112
113       // delete the pending commands that are already in the
              device_command table
114       const commandsToDeleteFromDeviceCommandTable = consecutiveCommands
              .filter(
115         (command) => !!command
116       )
117
118       // delete the commands from the device_command table
119       if (commandsToDeleteFromDeviceCommandTable.length > 0) {
120         await this.deviceCommandsRepository.delete(
121           commandsToDeleteFromDeviceCommandTable.map((command) =>
                  command.id)
122         )
123
124         returnObject['deviceCommandsDeleted'] =
125           commandsToDeleteFromDeviceCommandTable
126       }
127       returnObject['deviceCommandsCreated'] =
128         await this.deviceCommandsRepository.save(
129           deviceCommandsWithoutConsecutiveCommands
130         )
131     }
132
133     return returnObject
134   }
135
136   async hasBookingConflict(
137     fieldId: string,
138     startTime: string,
139     endTime: string
140   ): Promise<boolean> {
141     const conflict = await this.bookingsRepository.query(
142       `SELECT COUNT(*) > 0 AS has_overlap
143       FROM booking
144       WHERE "fieldId" = $1
145         AND "startTime" < $3
146         AND "endTime" > $2`,
147       [fieldId, startTime, endTime]
148     )
149     return conflict[0]?.has_overlap
150   }
```

Código 5.5 – Exemplo de *controller* para *login*.

5.2.3.8.2 Listagem de Reservas de uma Quadra

- **Rota:** `GET /tenants/:tenantId/fields/:fieldId/bookings`
- **Descrição:** Retorna todas as reservas vinculadas a uma quadra específica.
- **Fluxo:**
  1. O usuário faz uma requisição para listar as reservas de uma quadra específica.
  2. O serviço busca todas as reservas associadas à quadra no banco de dados.
  3. A lista das reservas cadastradas é retornada na resposta.

5.2.3.8.3 Listagem de Reservas Futuras por Período

- **Rota:** `POST /tenants/:tenantId/fields/:fieldId/bookings/future`
- **Descrição:** Retorna todas as reservas futuras dentro de um determinado período.
- **Fluxo:**
  1. O usuário faz uma requisição informando um intervalo de tempo desejado.
  2. O serviço busca todas as reservas futuras que se enquadram no período informado.
  3. A resposta retorna uma lista de reservas futuras.

5.2.3.8.4 Consulta de uma Reserva Específica

- **Rota:** `GET /tenants/:tenantId/fields/:fieldId/bookings/:id`
- **Descrição:** Buscar uma reserva específica dentro de uma quadra esportiva.
- **Fluxo:**
  1. O usuário faz uma requisição com o identificador da reserva e da quadra correspondente.
  2. O serviço consulta o banco de dados para localizar a reserva correspondente.
  3. Se encontrada, os detalhes da reserva são retornados na resposta.
  4. Se a reserva não for encontrada, uma exceção é lançada informando o erro.

5.2.3.8.5 Cancelamento de uma Reserva

- **Rota:** `DELETE /tenants/:tenantId/fields/:fieldId/bookings/:id`
- **Descrição:** Cancela uma reserva existente e remove os comandos IoT associados.
- **Fluxo:**
  1. O usuário autenticado faz uma requisição para cancelar uma reserva existente.
  2. O serviço verifica se a reserva existe.
  3. Caso existam comandos IoT pendentes para a reserva, estes são removidos do banco de dados.
  4. A reserva é marcada como cancelada e excluída do banco de dados.
  5. A resposta confirma o cancelamento da reserva e a remoção dos comandos IoT.

5.2.3.9 Envio de Comandos IoT

Visão Geral

O módulo *schedule* é responsável por garantir o envio dos comandos IoT para a API eWeLink no momento adequado. Para isso, ele utiliza a biblioteca `@nestjs/schedule`, que permite a execução de tarefas periódicas dentro do sistema. O agendamento periódico verifica comandos pendentes e os envia à API eWeLink, garantindo que os dispositivos IoT sejam acionados conforme as reservas criadas no sistema.

O processo de agendamento é realizado com a anotação `@Cron('* * * * *')`, que configura a execução da tarefa a cada minuto. Durante essa execução, o sistema:

- Consulta os comandos pendentes no banco de dados.
- Agrupa os comandos por *tenant*.
- Filtra e prioriza os comandos mais recentes para cada dispositivo.
- Envia os comandos selecionados à API eWeLink.
- Atualiza o status dos comandos no banco de dados.

Fluxo de Execução

O módulo segue um fluxo detalhado para processar os comandos de dispositivos IoT de maneira eficiente e segura:

1. Consulta de Comandos Pendentes

- O sistema busca no banco de dados os comandos cujo horário de execução já tenha sido alcançado e que ainda estejam no estado `PENDING`.
- Os comandos são agrupados por *tenant* para garantir que cada empresa tenha seus dispositivos controlados separadamente e com uma única chamada à API EWeLink.

2. Filtragem de Comandos Duplicados

   - Para cada dispositivo IoT, apenas o comando mais recente é mantido para execução, evitando envio de comandos desnecessários.
   - Comandos mais antigos para o mesmo dispositivo são marcados como `SKIPPED` para evitar acionamentos desnecessários.

3. Envio dos Comandos à API eWeLink

   - Os comandos selecionados são formatados no padrão aceito pela API eWeLink demonstrado no Código 5.6.
   - O sistema busca e verifica se os *access tokens* de cada empresa estão válidos, caso contrário utiliza os *refresh tokens* para obter um novo *access token* ou emite um erro caso o *refresh token* esteja expirado e continua para os próximos *tenants*.
   - O sistema envia os comandos em lote para otimizar a comunicação.
   - A resposta da API é analisada para verificar quais comandos foram executados com sucesso e quais falharam.

```
1  [
2    {
3      type: 0 | 1,
4      id: "<id do dispositivo na plataforma ewelink>",
5      params: {
6        switch: "on" | "off"
7      }
8    }
9  ]
```

Código 5.6 – Formato de envio de comando para API.

O serviço desenvolvido para envio dos comandos à API eWeLink com auxílio da biblioteca *ewelink-api-next* desenvolvida pela *Coolkit* na linguagem javascript está disponível no Código 5.7.

```
1  async controlBatchThing(tenantId: string, thingList:
       SetThingDto[]) {
2  const { accessToken, region } =
```

```
3       await this.checkAccessAndRefreshToken(tenantId)
4
5     this.client.at = accessToken
6     this.client.region = region
7     this.client.setUrl(region)
8
9     const response = await this.client.device.setAllThingStatus({
          thingList })
10
11    return response
12  }
```

Código 5.7 – Serviço para envio de comandos IoT à API eWeLink.

4. Atualização do Status dos Comandos

   - Os comandos que foram executados com sucesso são marcados como `EXECUTED`.
   - Os comandos que falharam na execução são marcados como `FAILED`.
   - Em caso de erro geral na execução, todos os comandos do *tenant* são atualizados para `FAILED`.

Conclusão

O módulo *schedule* desempenha um papel fundamental na automação dos dispositivos IoT integrados ao sistema. Ao utilizar a biblioteca `@nestjs/schedule`, garante-se que os comandos sejam processados no momento correto, evitando acionamentos incorretos e otimizando a interação com a API eWeLink. Essa abordagem melhora a eficiência operacional do sistema e proporciona uma experiência mais confiável para o envio de comandos ao dispositivos IoT.

5.3 ANÁLISE DE RESULTADOS

Este capítulo apresenta uma análise detalhada dos resultados obtidos com a implementação do sistema de gerenciamento de reservas de quadras esportivas e automação de dispositivos IoT. A avaliação se baseia em indicadores de desempenho, gráficos e estatísticas para determinar a eficácia da solução desenvolvida em comparação com métodos anteriores.

5.3.1 Avaliação dos Resultados

A implementação realizada solucionou o problema tratado ao fornecer uma aplicação de servidor eficiente para o agendamento de quadras esportivas e o controle automatizado de dispositivos IoT com fácil configuração através da integração com a

plataforma EWeLink. Os resultados obtidos foram analisados a partir dos seguintes aspectos:

5.3.1.1 Eficiência e Desempenho

O novo sistema demonstrou desempenho significativo no processo de reservas e na automação de dispositivos. Através de uma bateria de testes realizados através de um script automatizado para envio de diversas requisições, foi observado que o sistema foi capaz de manter velocidade média de resposta das requisições inferior à 2 segundos para as principais funcionalidades analisadas em um ambiente de testes local com hardware de nível intermediário presente na Tabela 3. As chamadas foram realizadas de um outro computador através da rede local, a fim de simular um ambiente real.

Tabela 3 – Hardware de testes utilizado na análise de desempenho do sitema.

Dispositivo	**marca**	**modelo**
Processador	AMD	Ryzen 5 3600
Placa-mãe	Gigabyte	B450M DS3H V2
Placa de vídeo	Gigabyte	RTX3070 Aorus Master
Memória RAM	XPG	4x8GB DDR4 3000MHz
SSD	KingDian	512GB NVME read/write speed 2500MB/s | 1500MB/s

Fonte: Autor.

O código base desenvolvido para realiza n chamadas ao endpoit e calcular o tempo médio de resposta está apresentado no Código 5.8.

```
1  const axios = require('axios');
2
3  const sendRequest = (startTime, i) => {
4    const timePromise = new Promise(resolve => {
5        axios.get(url, {headers: headerWithAccessToken});
6        const responseTime = Date.now() - startTime;
7        console.log(`Request ${i + 1} took ${responseTime} ms`)
8        resolve(responseTime)
9    })
10   return timePromise
11 }
12
13 const headerWithAccessToken = {
14   headers: {
15     'Authorization': `Bearer ACCESS_TOKEN`
16   }
17 }
18
```

```
19  async function measureAverageResponseTime(url, n) {
20    let promises = []
21
22    for (let i = 0; i < n; i++) {
23      const start = Date.now();
24      try {
25        const timePromise = sendRequest(start, i)
26        promises.push(timePromise)
27      } catch (error) {
28        console.error(`Request ${i + 1} failed:`, error.message);
29      }
30    }
31    const resolvedTimes = await Promise.all(promises)
32    const totalTime = resolvedTimes.reduce((acc, curr) => acc + curr, 0)
33
34    const averageResponseTime = totalTime / n;
35    console.log(`Average response time over ${n} requests: ${
        averageResponseTime.toFixed(0)} ms`);
36  }
37
38  // Example usage:
39  const apiUrl = 'http://localhost:3000/tenants/451e6db3-1a94-4918-9ec1-
      b8aa793f0269/fields/5b7a652f-0ddb-4a52-b552-4f92a02026d2/bookings';
       // To get all bookings from a field
40  const numberOfCalls = 1000;
41
42  measureAverageResponseTime(apiUrl, numberOfCalls);
```

Código 5.8 – *script* para cálculo do tempo médio de resposta.

Para alguns *endpoints* foi necessário realizas algumas modificações no código para garantir que ele funcionasse corretamente. Por exemplo, alguns endpoints exigiam autenticação e outros não. Além disso, alguns endpoints exigiam parâmetros específicos na requisição, como e-mails, senhas e outros dados, sendo necessário criar listas com parâmetros *mockados*. Com os *scripts* de teste prontos, foi possível realizar a análise dos principais endpoints do sistema realizando 1000 chamadas para cada e calcular o tempo médio de resposta.

A Tabela 4 apresenta o tempo médio de resposta para cada um dos principais e mais críticos endpoints da API. Esses resultados comprovam a eficiência e desempenho do novo sistema em gerenciar reservas de quadras e controlar dispositivos IoT, demonstrando sua capacidade de atender às necessidades dos usuários e melhorar a experiência de locação, além de atender os requisitos técnicos desejados e definidos no início do projeto. Apesar do ambiente testado ter sido local, acredita-se que o sistema será capaz de seguir os requisitos mesmo quando estiver executando em um servidor remoto, pois houve uma boa margem de segurança do resquisito técnico

Tabela 4 – Tempo médio de resposta para os principais *endpoints* analisados.

Enpoint	**método**	**descrição**	**tempo (ms)**
/auth/login	POST	login de usuário	12
/auth/register	POST	Registra um novo usuario	174
/auth/refresh	GET	Renova o token de acesso	52
/users	GET	Lista todos os usuarios	204
/tenants	GET	Lista todos os tenants	124
/tenants/:tenantId/fields			
/	GET	Lista todas as quadras de um tenant	155
/:id	GET	Retorna os detalhes de uma quadra especifica	73
/:fieldId/field-availabilities	GET	Lista as disponibilidades de uma quadra	212
/:fieldId/bookings	POST	Cria uma nova reserva	230
/:fieldId/bookings	GET	Lista todas as reservas de uma quadra	344
/:fieldId/bookings/future	POST	Lista reservas futuras	281
/:fieldId/bookings/future-by-user	POST	Lista reservas futuras por usuario	292
/:fieldId/bookings/:id	GET	Retorna os detalhes de uma reserva especifica	125
/:fieldId/field-devices	GET	Lista todos os dispositivos de uma quadra	123
/:fieldId/field-devices/:id	GET	Retorna os detalhes de um dispositivo	89

Fonte: Autor.

desejado em relação ao tempo médio das chamadas obtido.

5.3.2 Vantagens e Desvantagens da Solução Desenvolvida

Vantagens

- Integração direta com a API eWeLink para controle de dispositivos IoT.
- Simplicidade na aquisição, instalação, configuração e controle dos dispositivos IoT.
- Gestão eficiente de múltiplos *tenants* em um ambiente unificado.

- Redução do consumo de energia elétrica com acionamento automatizado dos dispositivos apenas nos horários de uso.
- Segurança aprimorada por meio de autenticação baseada em tokens e controle de permissões de acesso baseado em papéis.

Desvantagens

- Dependência da conectividade com a API eWeLink para a operação dos dispositivos IoT.
- Possível curva de aprendizado para novos usuários e administradores do sistema.
- Necessidade de manutenção e melhorias contínuas para atualização necessidades das empresas.

5.3.3 Problemas Encontrados

Durante a implementação, alguns desafios técnicos foram identificados, como a necessidade de lidar com comandos IoT duplicados ou atrasados e a otimização do processamento de comandos para evitar sobrecarga no sistema. Além disso, questões relacionadas à conectividade da API eWeLink impactaram levemente na confiabilidade da automação dos dispositivos pois, em alguns momentos, a API pode ficar indisponível ou apresentar lentidão no envio dos comandos aos dispositivos. Embora essa lentidão ocorra, o impacto não deve ser significativo, pois em nenhum caso o envio de comandos demorou mais de 20 segundos para ser efetivado.

5.3.4 Impacto dos Resultados Obtidos

Através de testes simulados, notou-se que a futura implementação do sistema completo trará impactos significativos nos processos e serviços das empresas clientes, com destaque para os seguintes aspectos:

Impactos Organizacionais

- Maior controle sobre a gestão de reservas e utilização dos espaços esportivos.
- Melhoria na comunicação com clientes devido à transparência na disponibilidade de horários.

Impactos Tecnológicos

- Utilização de tecnologias modernas como *NestJS*, *PostgreSQL* e *TypeORM*.

- Integração bem-sucedida com dispositivos IoT para automação.

Impactos Financeiros

- Redução de custos operacionais com a automação do acionamento de dispositivos elétricos.
- Potencial aumento da receita devido à otimização do uso das quadras.

Impactos Ecológicos

- Redução do consumo de energia elétrica por meio da automação dos dispositivos IoT.
- Contribuição para práticas mais sustentáveis no gerenciamento dos espaços esportivos.

Conclusão

A análise dos resultados demonstra que o sistema backend desenvolvido atendeu aos objetivos propostos e trouxe melhorias significativas para a gestão de reservas e automação de dispositivos IoT em ambiente simulado. Apesar dos desafios encontrados, a implementação se mostrou eficaz e com grande potencial de expansão e aprimoramento futuro.

6 CONCLUSÃO

Instruções da Coordenação do PFC:
A Conclusão deve apresentar:

- Uma **síntese** do problema tratado, da solução proposta e dos principais resultados obtidos;
- Uma discussão sobre o que foi atingido no PFC em relação aos objetivos inicialmente traçados e sobre as limitações encontradas;
- Uma discussão sobre possíveis aprimoramentos;
- Destacar os impactos do PFC para a empresa/clientes da empresa/instituto de pesquisa, além de possíveis impactos organizacionais, tecnológicos, financeiros, éticos, ecológicos, etc.
- Sugestão de trabalhos futuros (como se poderia dar continuidade ao PFC; aplicar o desenvolvimento realizado no PFC a outros problemas/processos; etc)

REFERÊNCIAS

ERL, Thomas; PUTTINI, Ricardo; MAHMOOD, Zaigham. **Cloud Computing: Concepts, Technology & Architecture**. [S.l.]: Pearson, 2013. p. 528. ISBN 9780133387520.

FIELDING, Roy T. Architectural styles and the design of network-based software architectures. UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

FOWLER, M. **Patterns of Enterprise Application Architecture**. [S.l.]: Addison-Wesley, 2002.

GOKHALE, Pradyumna; BHAT, Omkar; BHAT, Sagar. Introduction to IOT. **International Advanced Research Journal in Science, Engineering and Technology**, v. 5, n. 1, p. 41–44, 2018.

MARTIN, Robert C. Design principles and design patterns. **Object Mentor**, v. 1, n. 34, p. 597, 2000.

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [S.l.]: Pearson, 2017. ISBN 0134494164.

MASSE, Mark. **REST API design rulebook**. [S.l.]: "O'Reilly Media, Inc.", 2011.

SHENZHEN COOLKIT TECHNOLOGY CO., LTD. **OAuth2.0**. Disponível em: https://coolkit-technologies.github.io/eWeLink-API/#/en/OAuth2.0. Acesso em: 23 nov. 2024.

TURNER, M.; BUDGEN, D.; BRERETON, P. Turning software into a service. **Computer**, v. 36, n. 10, p. 38–44, 2003. DOI: 10.1109/MC.2003.1236470.