UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Vinícius Limas Scheidt

Software para Registros de Inspeção de Processos Digitais

Blumenau
2024

Vinícius Limas Scheidt

Software para Registros de Inspeção de Processos Digitais

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheiro de Controle e Automação.
Orientador: Prof. Orientador, Dr. Mauri Ferrandin

Blumenau
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Scheidt, Vinícius Limas
Software para Registros de Inspeção de Processos Digitais / Vinícius Limas Scheidt ; orientador, Mauri Ferrandin, 2024.
58 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Graduação em Engenharia de Controle e Automação, Blumenau, 2024.

Inclui referências.

1. Engenharia de Controle e Automação. 2. digitalização de processos. 3. normalização de dados. 4. inspeção de processo. I. Ferrandin, Mauri. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Vinícius Limas Scheidt

Software para Registros de Inspeção de Processos Digitais

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Engenheiro de Controle e Automação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 18 de Dezembro de 2024.

Banca Examinadora:

Documento assinado digitalmente
Mauri Ferrandin
Data: 20/12/2024 15:34:39-0300
CPF: ***.580.869-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Mauri Ferrandin, Dr.
Universidade Federal de Santa Catarina - Campus Blumenau

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 20/12/2024 15:38:16-0300
CPF: ***.999.369-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Carlos Roberto Moratelli, Dr.
Universidade Federal de Santa Catarina - Campus Blumenau

Documento assinado digitalmente
Maiquel de Brito
Data: 20/12/2024 15:38:30-0300
CPF: ***.657.150-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Maiquel de Brito, Dr.
Universidade Federal de Santa Catarina - Campus Blumenau

Este trabalho é dedicado a meus pais, familiares, amigos, professores e todos aqueles que me apoiaram durante a graduação.

AGRADECIMENTOS

Agradeço a minha família por ser meu exemplo e me apoiar. Agradeço aos professores que contribuíram para a minha formação, e enalteço meu orientador por seu auxílio e prontidão. Também agradeço meus amigos pelos bons momentos compartilhados. Ademais, gostaria de agradecer a Bosch Rexroth pela confiança e meus colegas de trabalho por todo o suporte para desenvolver esse projeto. Também agradeço à Universidade Federal de Santa Catarina, Campus Blumenau, pela notável excelência no ensino oferecido.

RESUMO

A digitalização na indústria moderna reflete a busca por maior eficiência, rastreabilidade e qualidade nos processos produtivos. Empresas enfrentam o desafio de integrar tecnologias para substituir métodos manuais e descentralizados, de modo a assegurar controle e confiabilidade nas operações. Esse cenário é especialmente relevante em ambientes industriais que demandam precisão no registro e na análise de dados, bem como a automação de processos para suportar decisões ágeis e fundamentadas. Para isto, este trabalho consiste em uma solução computacional para a digitalização de processos industriais, que inclui armazenamento e gerenciamento de dados em uma base relacional estruturada conforme os princípios da normalização. Ademais, utiliza-se o ASP.NET Core e o Entity Framework Core para implementar uma API RESTful que realiza integração com outros componentes. A arquitetura do sistema abrange componentes para processamento automatizado de dados, gestão de tabelas relacionais e interface gráfica interativa em Windows Forms, de modo a otimizar a coleta, validação e análise de dados de inspeção da empresa.

Palavras-chave: digitalização de processos; normalização de dados; inspeção de processo

ABSTRACT

Digitization in the modern industry reflects the pursuit of greater efficiency, traceability, and quality in production processes. Companies face the challenge of integrating technologies to replace manual and decentralized methods, ensuring control and reliability in operations. This scenario is particularly relevant in industrial environments that demand precision in data recording and analysis, as well as process automation to support agile and well-informed decision-making. For this purpose, the system consists of a computational solution for the digitization of industrial processes, including data storage and management in a relational database structured according to normalization principles. Moreover, ASP.NET Core and Entity Framework Core are used to implement a RESTful API that integrates with other components. The system architecture encompasses components for automated data processing, relational table management, and an interactive graphical interface in Windows Forms, aiming to optimize the collection, validation, and analysis of the company's inspection data.

Keywords: digitalization; data normalization; process inspection

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

No cenário industrial contemporâneo, a digitalização de informações e processos emerge como requisito indispensável para a evolução e eficiência de empresas que almejam se manter competitivas frente às rápidas transformações tecnológicas do mercado. Esse movimento compreende a aplicação de tecnologias emergentes para otimizar fluxos produtivos, reduzir erros, melhorar a eficiência e aumentar a rastreabilidade dos dados de produção. Dentre as tecnologias-chave desse conceito estão a automação, a conectividade entre dispositivos e a análise de dados em tempo real, que oferecem às empresas um controle mais preciso de seus processos (LIMA; GOMES, 2020). A adoção dessas práticas possibilita ambientes de produção onde os sistemas sejam capacitados a responder de forma autônoma às mudanças, de modo a disponibilizar informações que fundamentem a tomada de decisão em tempo hábil e provendo gerenciamento robusto da cadeia produtiva (SENSIO, 2024).

1.1 A EMPRESA

A Bosch Rexroth é uma empresa que integra o grupo Bosch e é uma das líderes globais no fornecimento de tecnologias de acionamento e controle para a indústria, com ênfase na fabricação de sistemas hidráulicos para aplicações móveis e industriais. Na unidade localizada em Pomerode, Santa Catarina, a empresa atua na produção de componentes voltados a esses dois segmentos, com uma linha de produtos que inclui bombas, motores e blocos de comando para sistemas hidráulicos (Figura 1) (LINSINGEN, 2016). Esses produtos são amplamente utilizados em setores que exigem precisão, durabilidade e confiabilidade, como a indústria automotiva, de construção, de mineração e agrícola.

Visando à excelência, a Bosch Rexroth investe continuamente em iniciativas voltadas à digitalização, com o desenvolvimento de projetos internos que alinham inovação e eficiência. Um exemplo notável é o “Fórum de Talentos”, evento anual promovido pela empresa, onde participantes do programa de estágio têm oportunidade de apresentar projetos/protótipos de diversas áreas. Esses projetos, habitualmente, contemplam conceitos e premissas da Indústria 4.0, que incluem a automação, a conectividade e a análise de dados para otimizar processos e aumentar a eficiência.

1.2 REGISTRO DE INSPEÇÃO DE PROCESSO (RIP)

O processo de inspeção de um produto consiste na verificação detalhada de características e especificações do item, afim de assegurar padrões de qualidade estabelecidos, que comumente compreende o uso de ferramenta(s) especializada(s). Para isso, o Registro de Inspeção de Processo (RIP) é um conceito de documento de controle de qualidade amplamente utilizado na indústria (ZAKLOUTA, 2011). No contexto deste trabalho, é

Figura 1 – Bosch Rexroth - Planta de Pomerode.

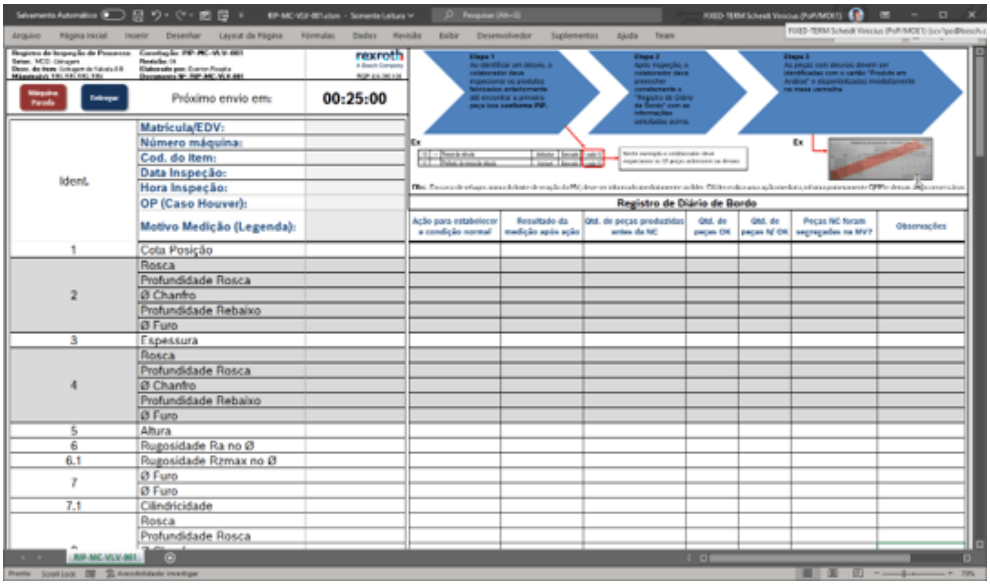
Fonte: Divulgação / Bosch Rexroth

um padrão de documento de responsabilidade da Engenharia de Processos, designado por uma lista de cotas com identificadores únicos, descrições, e campos para preenchimento de operadores de usinagem/montagem (de acordo com a área do documento). Cada RIP incorpora todas as cotas presentes em um Plano de Inspeção de Processo (PIP), documento que define os elementos de inspeção de um produto. Para cada elemento, são definidos: valor nominal, tolerância, metodologia (atributiva ou variável), frequência de registro, sistema de medição, entre outras informações.

Por se tratar de um método altamente manual e arcaico para realização e armazenamento de registros de inspeção, uma vez que os mesmos eram mantidos em cópias de papel, a primeira instância da digitalização dos RIPs foi implementada para garantir eficiência e rastreabilidade dos processos de inspeção de qualidade na Bosch Rexroth. O projeto foi elaborado e executado no decorrer do segundo semestre de 2023. A aplicação, apresentada na Figura 2, foi desenvolvida em VBA Excel, e publicada na plataforma integrada de gestão de documentos da empresa (ISODOC, desenvolvida pela *SoftExpert Suite*), como uma estrutura homologada *.xlsb*, que contém subrotinas e macros que automatizam e padronizam o registro das inspeções.

O sistema proposto neste trabalho consiste em uma aplicação dedicada, por meio de *software*, em conjunto com um *worker service* (MICROSOFT LEARN, 2024e) e uma base de dados relacional, abordados no Capítulo 2 que detêm as funcionalidades primárias presentes na primeira instância do projeto, introduz novos elementos de autenticação, assegura a integridade de dados armazenados e aprimora a escalabilidade da ferramenta. Na seção a seguir, são descritas as funcionalidades da primeira instância mencionada.

Figura 2 – RIP digital: Primeira instância.



Fonte: Autor

1.2.1 Preenchimento automático e travamento de data/hora de submissão

Uma das funcionalidades primordiais do RIP digital é o preenchimento automático da data/hora de submissão do registro. Desenvolvido por meio de manipuladores de eventos em VBA, esse recurso assegura que cada registro receba um carimbo temporal preciso e imutável. Após o início do preenchimento, o campo de data/hora é automaticamente bloqueado, assegurando que as informações registradas permaneçam inalteradas e que a rastreabilidade do processo seja mantida de maneira rigorosa.

1.2.2 Temporizador parametrizável para acompanhamento de processo

Como única subrotina executada em tempo real, consta-se o temporizador parametrizável, que proporciona monitoramento dos envios de inspeção, conforme o ciclo específico de cada processo, como usinagem ou brunimento. Este temporizador é definido pelo engenheiro responsável pelo processo e certifica que o RIP seja submetido para a base de dados em intervalos regulares, sem depender do preenchimento do registro. O temporizador provê, assim, um controle dinâmico do tempo de execução das operações, auxiliando na identificação de possíveis gargalos e desvios de produtividade.

1.2.3 Notificação automática de falta de registros

Com intuito de assegurar a conformidade e a pontualidade dos registros, foi implementada a funcionalidade de notificação automática via e-mail, para cenários de falta de submissão do RIP dentro de intervalos estipulados. A notificação, enviada aos líderes de

produção, inclui detalhes como equipamento, processo e tempo decorrido desde a última submissão, permitindo uma gestão proativa da produção. Esse recurso é primordial para a continuidade e a estabilidade das operações, uma vez que a execução destas atividades é intrinsecamente dependente de atuação humana.

1.2.4 Registro dinâmico e diário de bordo

O RIP digital detém um segmento denominado "diário de bordo", destinado a registros de informações adicionais sobre não conformidades nos processos (KUMAR; AZAD, 2017). Tais informações, que contemplam desde condições específicas de máquinas/equipamentos até observações sobre a qualidade do material, são automaticamente incorporadas à base de dados do RIP, de modo a possibiltar que usuários realizem análises completas e contextualizadas sobre as operações. Esse registro detalhado auxilia na identificação de padrões de falha e na implementação de melhorias nos processos de produção.

1.2.5 Centralização e criação de tabelas dedicadas para cada revisão do RIP

Para assegurar-se da preservação e o acesso à diferentes versões de um RIP, o sistema digital detém funcionalidade para geração de novas base de dados *.xlsx* dedicadas a cada revisão do documento. Esse recurso, apesar de não otimizado para escalabilidade do sistema, facilita o gerenciamento das revisões e possibilita que cada versão do documento esteja disponível para consulta, sem interferência de revisão(ões) posterior(es). A centralização dos registros em formato digital, além de simplificar o preenchimento, extingue a necessidade de cópias impressas, o que favorece a sustentabilidade e o aumento na produtividade.

1.2.6 Proteção por senha e modo somente leitura

Para prover uma camada de segurança da integridade de informações inseridas no RIP, o projeto do documento digital foi protegido por uma senha parametrizável, definida com base em sua respectiva área de implementação, e configurado para ser acessado em modo "somente leitura". A restrição previne alterações indesejáveis e indica que apenas usuários autorizados possam configurar parâmetros do RIP. Contudo, esses recursos, embora benéficos, não proporcionam níveis adequados de segurança contra possíveis violações ou acessos não autorizados, ressaltando a importância de práticas adicionais de segurança da informação.

1.3 OBJETIVOS

Nesta seção são descritos os objetivos geral e específicos do Trabalho de Conclusão de Curso.

1.3.1 Objetivo geral

Em consideração às áreas de melhoria identificadas na primeira instância do projeto, o objetivo geral do trabalho consiste em desenvolver uma solução via *software* que reestruture as principais funcionalidades do projeto inicial, de controle de RIPs, com base em práticas de desenvolvimento mais eficientes, contemple funções de segurança e autenticação adicionais, reduza o volume de documentos redundantes no sistema de gestão de documentos da empresa, e honre as normas de desenvolvimento e prototipagem de *software* da mesma.

1.3.2 Objetivos específicos

- **Desenvolver interface gráfica de formulário para sistemas Windows:** Em virtude ao compromisso institucional da utilização de sistemas operacionais da Microsoft, a aplicação deve ser desenvolvida e otimizada para a plataforma .NET. Ademais, convém conter elementos gráficos intuitivos e autoexplicativos, devido à elevada rotação de colaboradores em diferentes postos de trabalho.
- **Criar base de dados relacional normalizada dos RIPs:** Para minimizar redundâncias e inconsistência de dados, prover maior eficiência no armazenamento (quando comparada com a utilização de tabelas Excel) e promover integridade referencial, opta-se pela implementação de uma base de dados SQL que possibilite a manipulação de metadados de documentos, revisões e registros de inspeção, com seus respectivos relacionamentos.
- **Desenvolver uma API CRUD:** Com intuito de realizar operações básicas de manipulação de dados (*Create*, *Read*, *Update*, *Delete*) em uma base SQL, uma API CRUD é essencial para intermediar interações entre a aplicação e a base.
- **Implementar um *Worker Service* para cadastro de PIPs:** A adoção de um *Worker Service* para monitoramento da inserção de novos PIPs em um diretório controlado no *drive* corporativo proporciona um método otimizado para substituir o cadastro de RIPs individuais no sistema de gestão de documentos da empresa, uma vez que estes são utilizados unicamente para preenchimento de medições, e são integráveis aos respectivos PIPs dentro da estrutura da base de dados.
- **Incorporar funcionalidades da primeira instância:** Como o projeto consiste no incremento da primeira instância do RIP digital, o mesmo deve herdar as características principais da primeira versão. O desenvolvimento de tais funcionalidades, assim como de aspectos do *front end* da aplicação, deve priorizar a utilização de bibliotecas nativas, quando disponíveis, de modo a otimizar o uso de recursos computacionais e prover segurança aprimorada do sistema.

A organização dos capítulos seguintes é constituída pela revisão teórica dos conceitos, ferramentas e entidades empregados na solução, pelo desenvolvimento detalhado de cada um dos quatro componentes do sistema, por uma análise dos resultados e efeitos positivos do projeto no contexto da empresa e, por fim, uma conclusão que sumariza os aspectos pertinentes do projeto.

2 REVISÃO TEÓRICA

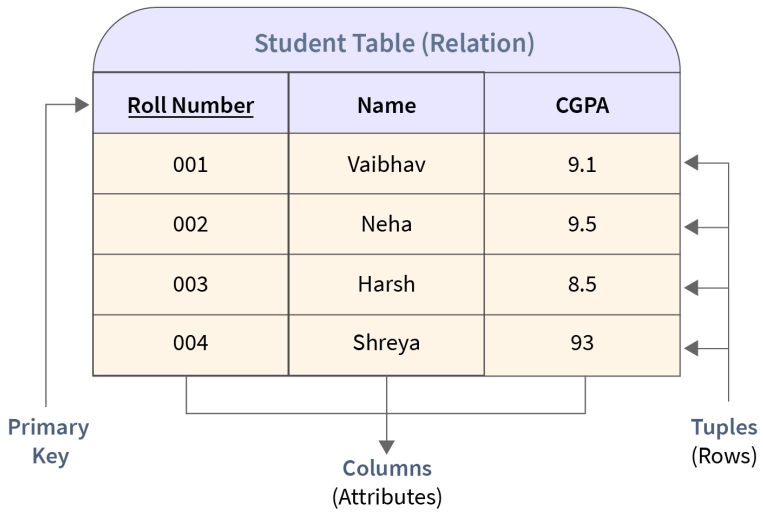
Neste capítulo são abordados os conceitos fundamentais e os elementos técnicos que sustentam o desenvolvimento do trabalho.

2.1 MODELO RELACIONAL E BASE DE DADOS RELACIONAL

O modelo relacional refere-se a uma estrutura conceitual para organizar dados em tabelas (ou relações) manipuláveis por operações matemáticas. De modo sucinto, descreve como os dados são organizados e as relações entre diferentes conjuntos de dados, ilustrado na Figura 3. A seguir, apresentam-se conceitos pertinentes à definição de modelo relacional:

- **Relação**: Uma tabela que organiza dados em linhas (tuplas) e colunas (atributos), representando uma coleção de informações estruturadas.
- **Tupla**: Uma linha de dados em uma relação, representando uma instância única de informações.
- **Atributo**: Uma coluna de dados em uma relação, representando uma característica ou propriedade de uma tupla.
- **Chave**: Um atributo ou conjunto de atributos que identifica de forma única uma tupla em uma relação. A chave primária é usada para garantir essa unicidade, enquanto chaves estrangeiras são usadas para criar relações entre diferentes tabelas. Chaves compostas são aplicáveis em casos de necessidade de definir uma tupla por mais de um atributo.

Figura 3 – Exemplo de relação e seus elementos



Fonte: (GULATI, 2024)

Propostos por Edgar F. Codd na década de 1970, os bancos de dados relacionais utilizam chaves primárias para identificar unicamente cada registro e chaves estrangeiras para estabelecer conexões lógicas entre tabelas, assegurando a integridade referencial (ORDONEZ; GARCÍA-GARCÍA, 2008). Além disso, os bancos de dados relacionais possibilitam a manipulação e recuperação eficiente dos dados por meio da linguagem SQL (*Structured Query Language*), amplamente adotada para realizar operações como inserção, consulta e modificação.

Os principais aspectos das bases de dados relacionais incluem:

- **Estrutura tabular**: Dados organizados em tabelas, otimizando o armazenamento e a consulta.
- **Integridade referencial**: Garantia de consistência entre os dados relacionados.
- **Flexibilidade**: Capacidade de adaptação a diferentes necessidades de armazenamento e consulta, devido à separação lógica entre dados e estrutura.

2.2 NORMALIZAÇÃO DE BASES DE DADOS RELACIONAIS

A *normalização de bases de dados relacionais* é o processo de organizar os dados em uma base de dados de modo a reduzir a redundância e eliminar anomalias estruturais (KUMAR; AZAD, 2017), capazes de surgir durante operações de inserção, atualização ou exclusão. O processo de normalização segue uma série de regras formais que resultam na decomposição de uma tabela em tabelas menores e mais eficientes, chamadas de *relações*, preservando a integridade e consistência dos dados.

O processo de normalização visa, sobretudo, estabelecer uma estrutura sólida e eficiente para o armazenamento e recuperação dos dados. Através da aplicação de formas normais, são eliminadas redundâncias e garantida a integridade referencial (ORDONEZ; GARCÍA-GARCÍA, 2008), isto é, a coerência dos dados em diferentes tabelas. Essa organização permite que o banco de dados opere de forma otimizada, evitando anomalias que possam comprometer a confiabilidade do sistema, como duplicações ou inconsistências entre registros.

Além de promover a integridade, a normalização facilita a manutenção e a expansão da base de dados ao longo do tempo, de modo a simplificar a adaptação a novas demandas sem impactar a consistência das informações já armazenadas. Embora o processo de normalização tenda a melhorar a eficiência lógica do sistema, o mesmo exige equilíbrio entre a normalização completa e a performance em consultas de grandes volumes de dados, especialmente em sistemas de tempo real, onde a desnormalização controlada pode ser aplicada em situações específicas.

2.2.1 Dependências Funcionais

O conceito de *dependência funcional* é central na normalização de dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2019). Uma dependência funcional estabelece uma relação entre dois conjuntos de atributos, onde o valor de um conjunto determina o valor do outro. Formalmente, uma dependência funcional é expressa como:

$$A \rightarrow B$$

onde A é o conjunto de atributos determinantes e B é o conjunto de atributos dependentes. Isso implica que, para cada valor de A, existe um único valor correspondente de B. Dependências funcionais são usadas para definir chaves primárias e outras restrições de integridade, e sua análise é essencial para a correta decomposição das tabelas durante o processo de normalização.

2.2.2 Definição das Formas Normais

A normalização é dividida em diversas etapas, chamadas de *formas normais* (CORONEL; MORRIS, 2019). Cada forma normal resolve um conjunto específico de problemas relacionados à redundância e dependência de dados, conforme descrito a seguir.

2.2.2.1 Primeira Forma Normal (1FN)

Uma tabela está na **Primeira Forma Normal** (1FN) quando atende ao seguinte requisito:

- Todos os atributos devem ser atômicos, o que significa que cada atributo contenha um único valor, indivisível.

Uma violação típica da 1FN ocorre quando um campo contém uma lista de valores, como uma coluna “telefone” armazenando múltiplos números em uma única célula. Nesse caso, deve-se dividir esses valores em registros separados ou criar uma tabela auxiliar para armazená-los.

2.2.2.2 Segunda Forma Normal (2FN)

Para que uma tabela esteja na **Segunda Forma Normal** (2FN), é necessário que:

- A tabela esteja em conformidade com a 1FN.
- Todos os atributos não-chave dependam totalmente da chave primária, ou seja, não deve haver dependência parcial.

A *dependência parcial* ocorre quando um atributo não-chave depende apenas de parte de uma chave composta, e não da chave primária completa. Por exemplo, se a chave primária de uma tabela for composta por dois atributos (A e B), qualquer atributo não-chave deve depender de ambos A e B, e não apenas de um deles.

Se uma dependência parcial for identificada, a tabela deve ser decomposta de forma que cada parte da chave primária tenha sua própria tabela, eliminando assim a redundância e garantindo que todos os atributos não-chave dependam exclusivamente da chave primária inteira.

2.2.2.3 Terceira Forma Normal (3FN)

Uma tabela está na **Terceira Forma Normal** (3FN) se:

- Estiver em conformidade com a 2FN.
- Não houver *dependência transitiva* entre os atributos não-chave e a chave primária.

A dependência transitiva ocorre quando um atributo não-chave depende de outro atributo não-chave, que por sua vez depende da chave primária. Para eliminar essa dependência, a tabela deve ser decomposta de forma que cada atributo não-chave dependa diretamente da chave primária.

Por exemplo, se A é a chave primária e B e C são atributos não-chave, uma dependência transitiva ocorre se $A \rightarrow B$ e $B \rightarrow C$, ou seja, C depende de A através de B. Nesse caso, deve-se criar uma nova tabela onde B se torna a chave primária de uma nova relação que contém C.

2.2.2.4 Forma Normal de Boyce-Codd (BCNF)

A **Forma Normal de Boyce-Codd** (BCNF) (CODD, 1974) é uma generalização mais rigorosa da 3FN. Ela exige que, para qualquer dependência funcional $A \rightarrow B$, o atributo A deve ser uma superchave (isto é, uma chave primária ou candidata).

A BCNF é necessária em casos especiais onde a 3FN não resolve completamente as anomalias. Embora uma tabela na 3FN elimine dependências parciais e transitivas, pode haver situações em que uma dependência funcional indevida ainda persista, mesmo que não envolva diretamente a chave primária. A BCNF trata especificamente desses casos, exigindo que todos os determinantes sejam superchaves.

2.2.3 Processo de Normalização

O processo de normalização compreende as etapas descritas abaixo:

1. **Identificação das Dependências Funcionais**: Avaliam-se as relações de dependência entre os atributos da tabela, definindo as chaves primárias e outras dependências funcionais.
2. **Decomposição das Tabelas**: Se uma tabela não satisfizer os requisitos de uma forma normal específica, ela é decomposta em duas ou mais tabelas menores, que respeitem essas regras.

3. **Preservação da Integridade Referencial**: Durante a decomposição, é necessário garantir que as relações entre as tabelas decompostas mantenham sua integridade. Isso é assegurado por meio do uso de chaves estrangeiras, que preservam a conexão lógica entre as tabelas.

Ao garantir as etapas do processo, a normalização assegura as características descritas no início da seção, como a eliminação de redundâncias, a preservação da integridade referencial e a consistência entre tabelas, além de simplificar a manutenção e adaptação da base de dados a novas necessidades. Apesar da existência de mais duas formas normais (4FN e 5FN), a normalização até a 3FN é, na maioria dos casos, um compromisso ideal entre a complexidade da estrutura, a integridade dos dados e a eficiência do desempenho. Formas normais mais avançadas são aplicadas podem ser aplicadas em cenários específicos que exigem uma eliminação mais rigorosa de redundâncias.

2.3 APIS RESTFUL NO ASP.NET CORE FRAMEWORK

As APIs RESTful (*Representational State Transfer*) são amplamente utilizadas para a criação de interfaces padronizadas que permitem a comunicação entre sistemas distintos. Fundamentadas em princípios arquiteturais definidos por Fielding (2000), elas utilizam requisições HTTP para expor e consumir recursos de forma eficiente e escalável. O framework *ASP.NET Core*, desenvolvido pela Microsoft, contempla um conjunto de ferramentas para o desenvolvimento de APIs RESTful (RICHARDSON; RUBY, 2007), de modo a possibilitar a implementação de soluções modernas e multiplataforma com foco em desempenho e flexibilidade. A arquitetura de uma aplicação *ASP.NET Core* genérica é ilustrado na Figura 4.

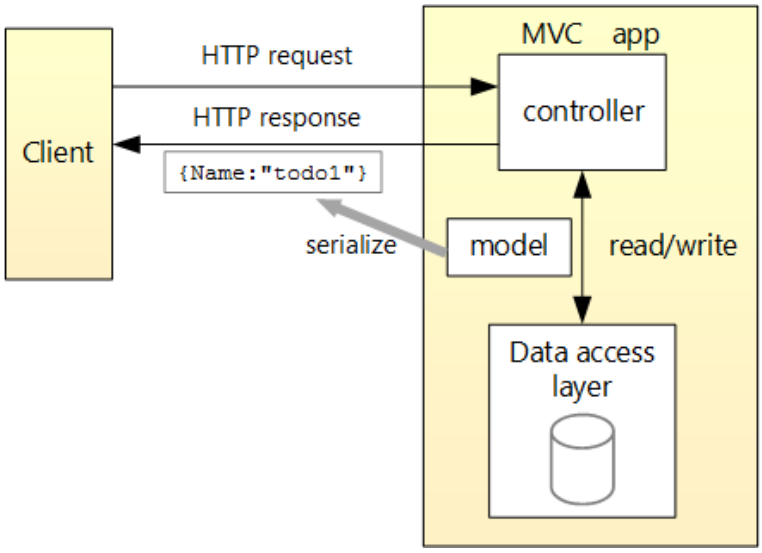
2.3.1 Protocolo HTTP

O protocolo HTTP (*Hypertext Transfer Protocol*) é fundamental para a comunicação na *web* e define um conjunto de regras para transferências de mensagens entre clientes e servidores. Tal protocolo opera com base em um modelo de requisição e resposta, onde o cliente (como um navegador *web*) faz uma solicitação ao servidor, que por sua vez processa essa solicitação e retorna uma resposta apropriada.

Para acessar recursos como texto, imagens, vídeos, etc., o cliente utiliza um URI (*Uniform Resource Identifier*) ou URL (*Uniform Resource Locator*), responsável pela identificação do local do recurso no servidor. Os métodos HTTP, como `GET`, `POST`, `PUT` e `DELETE`, definem o tipo de operação a ser realizada:

- `GET`: Solicita a recuperação de um recurso identificado pelo URI, como uma página HTML ou uma imagem.
- `POST`: Envia dados ao servidor para serem processados, como em formulários ou *uploads*.

Figura 4 – Arquitetura de uma aplicação *ASP.NET Core* genérica.



Fonte: (MICROSOFT LEARN, 2024f)

- `PUT`: Atualiza ou cria um recurso no servidor, no local especificado pelo URI.
- `DELETE`: Remove o recurso identificado pelo URI.

Quando o cliente faz uma requisição, o servidor retorna uma resposta contendo o código de status HTTP (como 200 para sucesso ou 404 para recurso não encontrado) e o corpo da resposta, que pode incluir o recurso solicitado, como um arquivo HTML, uma imagem, ou um vídeo.

O HTTP é um protocolo sem estado, de modo que cada requisição é independente e não mantém informações sobre requisições anteriores. Para gerenciar sessões ou estados, é comum utilizar cookies, tokens ou outras tecnologias adicionais. Essa característica proporciona flexibilidade e escalabilidade às aplicações distribuídas.

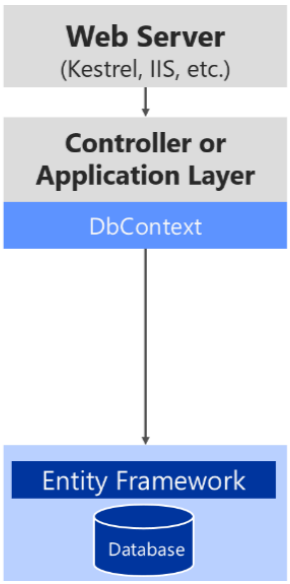
2.3.2 Configuração de Controladores

No *ASP.NET Core*, os controladores representam a camada responsável por gerenciar as requisições recebidas e fornecer as respostas apropriadas aos clientes. Estes controladores são definidos como classes que agrupam ações associadas a diferentes verbos HTTP. A configuração e o roteamento dessas ações são realizados por meio de atributos decoradores, os quais mapeiam as requisições para os métodos correspondentes. Essa abordagem modular e declarativa permite a organização sucinta do código, favorecendo a manutenção e a extensão da aplicação.

2.3.3 Entity Framework Core

O *Entity Framework Core (EF Core)* é um *Object-Relational Mapper* (ORM) integrado ao *ASP.NET Core* que abstrai o acesso a bases de dados relacionais (MICROSOFT LEARN, 2024a). Esse ORM possibilita que desenvolvedores interajam com dados por intermédio de classes e objetos em vez de comandos SQL, ilustrado pela camada do controlador na Figura 5, de modo a reduzir a complexidade do código e promover maior consistência lógica. Por meio do mapeamento entre as entidades do domínio e as tabelas da base de dados, o EF Core facilita operações como consultas, inserções e atualizações, assegurando que as relações e restrições de integridade sejam respeitadas. Adicionalmente, o *framework* oferece suporte para migrações de base de dados, o que simplifica a evolução do modelo de dados ao longo do ciclo de vida do sistema.

Figura 5 – Arquitetura de Aplicação com Entity Framework.



Fonte: (MICROSOFT LEARN, 2024b)

2.3.4 Pipeline de Middleware

O conceito de *middleware* refere-se a componentes de *software* responsáveis por interceptar e processar requisições HTTP antes que estas sejam encaminhadas aos controladores, definidos na Seção 2.3.2 (MICROSOFT LEARN, 2024d). No *ASP.NET Core*, o pipeline de middleware é configurável e possibilita a adição de funcionalidades como autenticação, autorização, compressão de respostas, tratamento de erros e registro de *logs*. Essa arquitetura modular é fundamental para atender às necessidades específicas de cada aplicação, garantindo a escalabilidade e a segurança do sistema. O pipeline é organizado

de forma sequencial, em que cada componente processa ou delega a requisição ao próximo, formando uma cadeia de responsabilidades.

2.4 ARQUITETURA MVC

A arquitetura *Model-View-Controller* (MVC) é um padrão amplamente utilizado no desenvolvimento de aplicações *software*, sobretudo em sistemas web, devido à sua capacidade de separar responsabilidades de forma clara e sucinta (QURESHI; SABIR, 2014). Esse padrão organiza o código em três componentes principais: *Model* (Modelo), *View* (Visão) e *Controller* (Controlador). Cada componente desempenha funções específicas no fluxo de dados e interações da aplicação. Essa segmentação visa aumentar a manutenibilidade, a escalabilidade e a testabilidade do sistema.

2.4.1 Componentes da Arquitetura MVC

2.4.1.1 Model (Modelo)

O componente *Model* representa a lógica de negócios e os dados da aplicação. Deste modo, é responsável por gerenciar o estado da aplicação e por interagir com fontes de dados externas, como bancos de dados ou serviços web. No contexto de um sistema de gerenciamento de dados, o *Model* inclui a definição das entidades, regras de validação, operações CRUD e demais aspectos relacionados ao domínio da aplicação.

No *ASP.NET Core*, o *Model* frequentemente utiliza o *Entity Framework Core* para mapear as entidades do domínio às tabelas do banco de dados, permitindo que as interações com os dados sejam realizadas por meio de objetos e coleções.

2.4.1.2 View (Visão)

A *View* é responsável pela apresentação dos dados para o usuário final. Ela renderiza as informações do *Model* em um formato compreensível, geralmente por meio de páginas HTML no caso de aplicações web. As *Views* não contêm lógica de negócios, mas podem incluir lógica de apresentação, como iteração sobre listas de dados para exibição ou aplicação de formatações específicas.

No *ASP.NET Core MVC*, as *Views* são implementadas através da linguagem *Razor*, que permite misturar código C# com HTML de forma clara e eficiente. A separação das *Views* do restante do sistema facilita a personalização e a adaptação da interface de usuário, sem impactar as funcionalidades subjacentes.

2.4.1.3 Controller (Controlador)

O componente *Controller* atua como intermediário entre o *Model* e a *View*, coordenando as interações entre eles. Dessa forma, é responsável por receber as requisições do

usuário, processá-las e retornar as respostas apropriadas. Tal lógica inclui a manipulação de dados no *Model*, o encaminhamento das informações processadas para a *View* e o controle do fluxo de execução da aplicação.

Os *Controllers* no *ASP.NET Core* são classes que contêm métodos de ação, cada um associado a um ou mais verbos HTTP, como `GET`, `POST`, `PUT` ou `DELETE`. Essa abordagem modular permite organizar o código em funções específicas, promovendo clareza e reutilização.

2.4.2 Ciclo de Vida de uma Requisição MVC

O ciclo de vida de uma requisição no padrão MVC segue etapas bem definidas, de modo a assegurar a separação de responsabilidades entre os componentes. Quando uma requisição HTTP chega à aplicação, o fluxo ocorre da seguinte maneira:

- **Recebimento da Requisição:** O *Routing*, mecanismo de tratamento de uma requisição HTTP, identifica o controlador e o método de ação apropriado com base no URL e no verbo HTTP da requisição. Essa etapa determina qual parte da aplicação será responsável por processar a solicitação.
- **Processamento no Controller:** O *Controller* processa a requisição, interagindo com o *Model* para realizar operações de negócios ou para obter dados necessários. A lógica de controle decide como os dados devem ser manipulados e preparados para envio à interface.
- **Atualização da View:** Os dados obtidos do *Model* são enviados à *View*, que os renderiza no formato adequado para o cliente, como HTML ou outros formatos, de acordo com o tipo de aplicação.
- **Retorno da Resposta:** A resposta gerada pela *View* é enviada ao cliente. Isso pode incluir uma página HTML, um arquivo JSON para APIs RESTful, ou outros formatos de saída compatíveis.

Essa organização modular permite que cada componente detenha responsabilidades específicas, assim, assegura-se clareza na lógica do sistema, modularidade no desenvolvimento e facilidade de manutenção ao longo do tempo.

2.4.3 MVC no contexto do *ASP.NET Core*

No *ASP.NET Core*, o padrão MVC é implementado de forma nativa e extensível. A integração com ferramentas como o *Entity Framework Core* e o suporte a injeção de dependências (HUDLI, S. R.; HUDLI, R. V., 2013) tornam o framework uma escolha robusta para aplicações modernas. Além disso, o suporte a *middleware* permite personalizar o ciclo de vida das requisições, complementando o fluxo MVC.

Tal combinação de recursos faz do *ASP.NET Core MVC* uma solução compatível para o desenvolvimento de aplicações web, uma vez que atende tanto às necessidades de desempenho e escalabilidade quanto à flexibilidade na organização do código.

2.5 WORKER SERVICE

Um *Worker Service* é uma aplicação ou componente projetado para executar tarefas de forma contínua ou recorrente em segundo plano, sem a necessidade de interação direta com o usuário. Esse tipo de serviço é amplamente utilizado para automatizar processos, monitorar eventos ou realizar atividades assíncronas em sistemas de software.

Worker Services são particularmente úteis em cenários onde a execução de uma tarefa precisa ser isolada do fluxo principal de uma aplicação ou quando é necessário gerenciar operações que demandam longos períodos de execução. Entre as funcionalidades usuais de um *Worker Service*, estão:

- Processamento de filas ou mensagens em sistemas distribuídos.
- Sincronização de dados entre sistemas ou bancos de dados.
- Execução de tarefas programadas, como backups ou relatórios.
- Monitoramento de recursos ou eventos, como *logs* ou alterações de estado em sistemas externos.

A arquitetura de um *Worker Service* inclui, frequentemente, um ciclo de vida bem definido, com etapas como inicialização, execução de tarefas, monitoramento e encerramento. Além disso, pode ser implementado de formas distintas, de modo a depender da linguagem de programação ou da plataforma utilizada. De modo geral, envolve um mecanismo para gerenciar o ciclo de vida da aplicação, como *loops* contínuos, agendamento de tarefas ou tratamento de eventos.

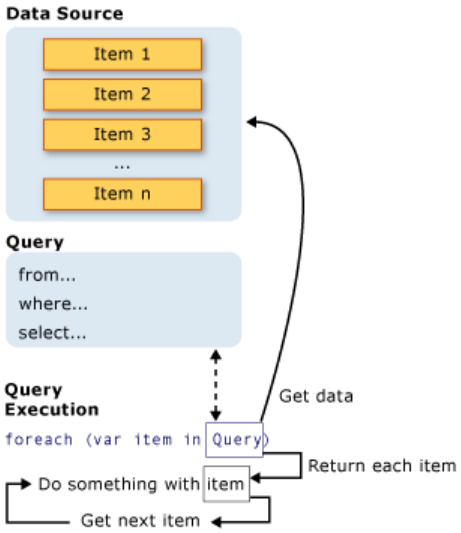
2.6 LINQ (LANGUAGE-INTEGRATED QUERY)

O *Language-Integrated Query* (LINQ) é uma linguagem de consulta a dados integrada ao *.NET Framework* que permite realizar consultas a diversas fontes de dados, como coleções em memória, arquivos XML e bases de dados, por meio da utilização de uma sintaxe declarativa e padronizada, que provê compatibilidade e precisão nos tipos de dados utilizados. (MICROSOFT LEARN, 2024c). Esta unifica a manipulação de dados ao oferecer uma interface consistente, baseada em expressões lambda e métodos de extensão. Contempla operações como filtragem, ordenação, agrupamento e projeção de forma eficiente e legível.

Conforme apresentado na Figura 6, estrutura do LINQ baseia-se em operadores padrão, como `Where`, `Select` e `OrderBy`, que implementam lógicas comuns de manipulação de dados. Essa abordagem promove segurança durante a compilação, previne erros típicos

de consultas dinâmicas, e auxilia na produtividade ao integrar a lógica de consultas diretamente ao código. Com suporte a diferentes implementações, como `LINQ to Objects` e `LINQ to XML`, trata-se de uma ferramenta versátil para o desenvolvimento moderno em .NET.

Figura 6 – Exemplo de utilização da LINQ.



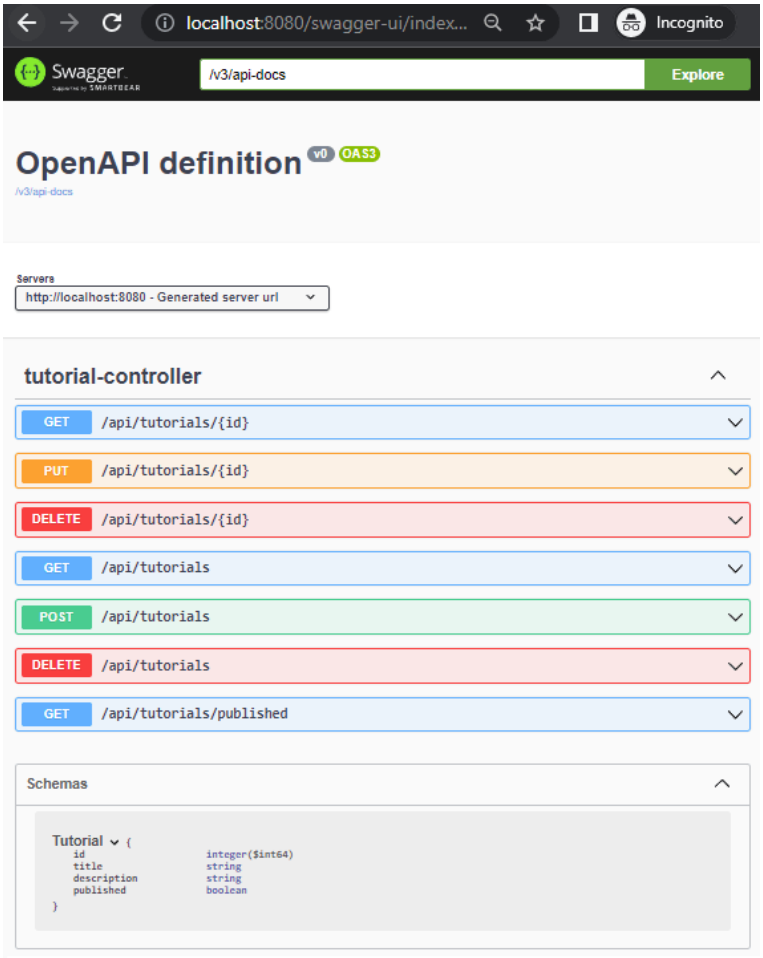
Fonte: (MICROSOFT LEARN, 2024c)

2.7 SWAGGER

A documentação é um aspecto fundamental no desenvolvimento de APIs RESTful, pois garante que os consumidores e desenvolvedores compreendam os recursos disponíveis e como interagir com os mesmos. Nesse contexto, o Swagger destaca-se como uma ferramenta para documentar, descrever e definir APIs de maneira estruturada, utilizando a especificação OpenAPI (SWAGGER, 2024). Essa especificação formalizada permite que as APIs sejam descritas de modo uniforme, possibilitando que diferentes ferramentas interpretem e apliquem as definições de maneira consistente.

Além de listar os *endpoints* e seus respectivos parâmetros, o Swagger fornece uma interface gráfica interativa que possibilita a execução de testes diretamente no navegador, como exemplificado na Figura 7. Essa funcionalidade não somente auxilia no desenvolvimento e validação de APIs, mas também permite que as definições sejam facilmente descobertas e compreendidas, tanto por usuários humanos quanto por outras aplicações, fortalecendo a integração e a comunicação entre as partes envolvidas.

Figura 7 – Exemplo de implementação do Swagger.



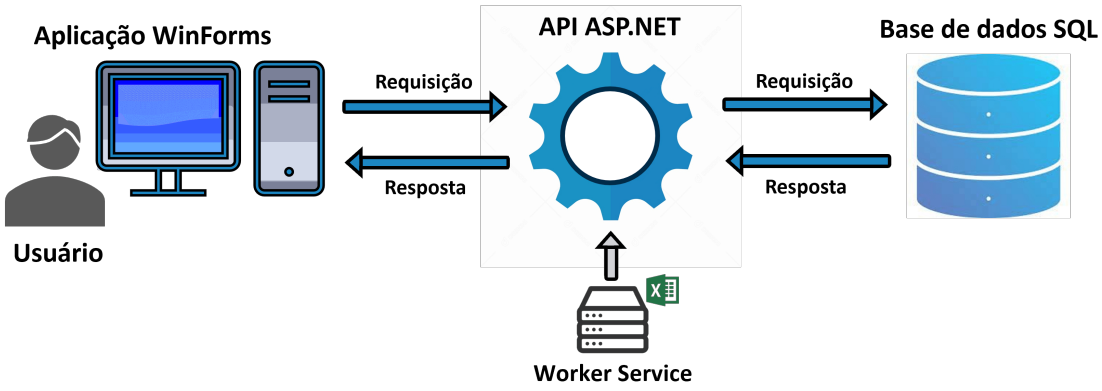
Fonte: (SWAGGER, 2024)

3 DESENVOLVIMENTO DO SISTEMA

Este capítulo almeja apresentar, de forma detalhada, a estrutura da base de dados e os quatro componentes da solução proposta. Sucintamente, os componentes são: um modelo da base de dados, que abstrai as entidades desta em forma de classes, na Seção 3.2; uma API, que atua como intermediária para acessar e manipular esses dados via operações RESTful, na Seção 3.3; um *Worker Service*, que automatiza a inserção de novos PIPs na base de dados ao monitorar diretórios e processar arquivos, na Seção 3.4; e um *Windows Forms*, que oferece uma interface gráfica para usuários visualizarem, registrarem inspeções e interagirem com o sistema de forma dinâmica e intuitiva, na Seção 3.5 .

Cada seção é dedicada para um item. O sistema foi integralmente desenvolvido no Visual Studio 2022, empregando a linguagem de programação C#, em um ambiente de desenvolvimento local que simula condições da intranet corporativa. Para ilustrar a estrutura macro do sistema e a interação entre os componentes, apresenta-se um diagrama na Figura 8.

Figura 8 – Diagrama de fluxo do sistema do RIP digital.

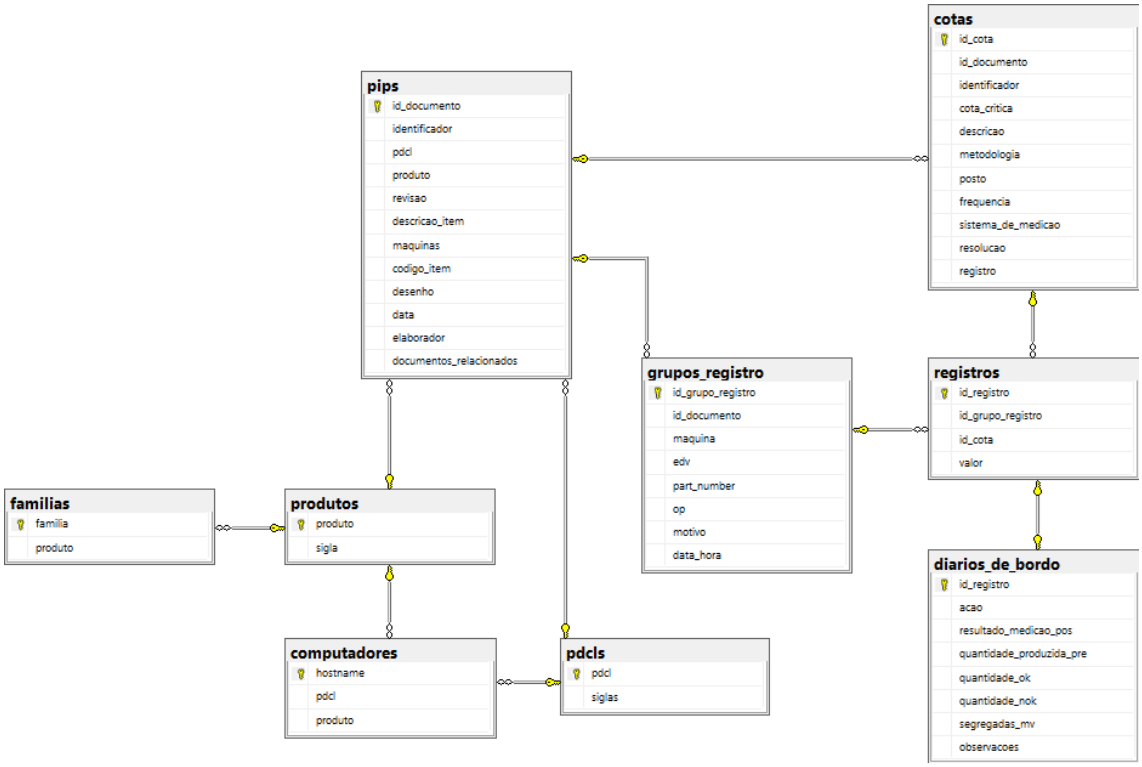


Fonte: Autor,2024

3.1 ESTRUTURA DA BASE DE DADOS RELACIONAL

A base de dados “Digitalização_MOE1” foi projetada com intuito de armazenar e gerenciar dados de iniciativas de digitalização de processos industriais do departamento. Para esse trabalho, princípios da normalização descritos na Seção 2.2 foram extensivamente aplicados nas tabelas que compõem o sistema, apresentadas no diagrama da Figura 9. A estrutura reflete uma organização modular, onde os dados são divididos em tabelas distintas, com tabelas interconectadas que garantem integridade referencial e minimizam redundâncias. A seguir, são detalhados os elementos principais da base, bem como seus relacionamentos e restrições.

Figura 9 – Diagrama da base de dados do projeto.



Fonte: Autor, 2024

3.1.1 Tabelas e Relacionamentos

A base de dados é composta por oito tabelas, cada uma desempenha um propósito único no gerenciamento das informações. Dentre as principais tabelas, têm-se:

3.1.1.1 Tabela `pips`

Armazena os metadados de documentos, os PIPs, descritos na Seção 1.2. Possui uma chave primária para cada documento (`id_documento`) e uma chave composta única (`identificador`, `revisao`) para assegurar versões distintas. A inclusão de identificadores únicos, para esta e para tabelas subsequentes, objetiva simplificar os relacionamentos entre as mesmas, auxiliar no desempenho em consultas e assegurar a integridade referencial, mesmo em casos de alterações em chaves compostas. Ademais, viabiliza a integração com futuras soluções da empresa e preserva a auditabilidade do sistema. Dessa forma, provê escalabilidade, flexibilidade e eficiência no gerenciamento dos dados.

Conexões: Relacionada a `cotas` através da chave estrangeira `id_documento`. Relacionada a `grupos_registro` pela mesma chave.

3.1.1.2 Tabela `cotas`

Gerencia o cadastro de cotas associadas aos PIPs. Possui chave primária (`id_cota`), e a chave composta única (`id_documento`, `identificador`) assegura caráter único de registro dentro da tabela. A composição de `cotas` com `pips` compreende todas as informações de um documento. A opção por dividi-las parte do princípio que revisões de um PIP e PIPs diferentes hão de contemplar cotas e/ou quantidades distintas, porém sempre serão definidas pelos mesmos campos de metadados.

Conexões: Relacionada a `registros` através da chave estrangeira `id_cota`.

3.1.1.3 Tabela `grupos_registro`

Trata-se da tabela que armazena medições realizadas. Dessa forma, compõe a parcela da base de dados destinada ao preenchimento efetivo dos RIPs. Em conjunto com a tabela `registros`, comporta o maior volume de dados gerados durante o uso da aplicação. Possui chave primária (`id_grupo_registro`). Possibilita rastreabilidade dos registros de inspeção através dos atributos máquinas, operadores (`edv`), e ordens de produção (`op`).

Conexão: Relacionada a `registros` através da chave estrangeira `id_grupo_registro`.

3.1.1.4 Tabela `registros`

Criada para armazenamento individual de medições de cotas, realizadas em cada campo de um RIP (apresentado na Seção 1.2). Possui chave primária (`id_registro`). De modo análogo à tabela `cotas`, a composição dessa tabela com `grupos_registro` contempla todas as informações de submissões de registros de inspeção de usuários da aplicação.

Conexões: Relacionada a `diarios_de_bordo` através da chave estrangeira `id_registro`.

3.1.1.5 Tabela `diarios_de_bordo`

Responsável por armazenar dados de medições não conformes, onde informações adicionais são requeridas junto com uma instância de `registros`, conforme descrito na Seção 1.2.4.

3.1.1.6 Demais tabelas

Com pertinência, porém não projetadas para definir características de um documento e suas respectivas medições, outras quatro tabelas de suporte são definidas.

- As tabelas `familias` e `produtos` são complementares, de modo que `familias` é utilizada para vincular a sigla de produtos, presente no campo “identificador” dos PIPs, às respectivas famílias do produto.

- A tabela `pdcls`, de forma similar, utiliza a sigla das áreas, denominadas PDCLs, presente no campo "identificador" dos PIPs, para vincular às respectivas áreas do ambiente de produção.
- A tabela `computadores` é fundamental para gerenciar informações sobre computadores de cada área. Através do *hostname*, variável do sistema Windows, provê acesso à seletos PIPs, com base no computador que executa a aplicação, de acordo com a área e produto que o mesmo está cadastrado.

3.1.2 Aspectos Técnicos das Configurações

Dentre os elementos técnicos relevantes da configuração da base, enfatiza-se o índice primário de cada tabela, que foi otimizado para suportar travamentos em nível de linha (`ALLOW_ROW_LOCKS`) e página (`ALLOW_PAGE_LOCKS`), de modo a prover controle de concorrência e maior eficiência no acesso aos dados durante operações simultâneas, uma vez que a aplicação é projetada para comportar múltiplas instâncias simultaneamente. Ademais, tem-se a configuração do índice primário com essas opções permite que bloqueios sejam aplicados de forma granular, para minimizar o impacto em outras transações que acessam a mesma tabela.

A base está configurada com verificação de integridade (`PAGE_VERIFY CHECKSUM`) e atualização automática de estatísticas (`AUTO_UPDATE_STATISTICS ON`), de modo a prover detecção de corrupção em páginas de dados e manutenção de estatísticas atualizadas para otimização de consultas.

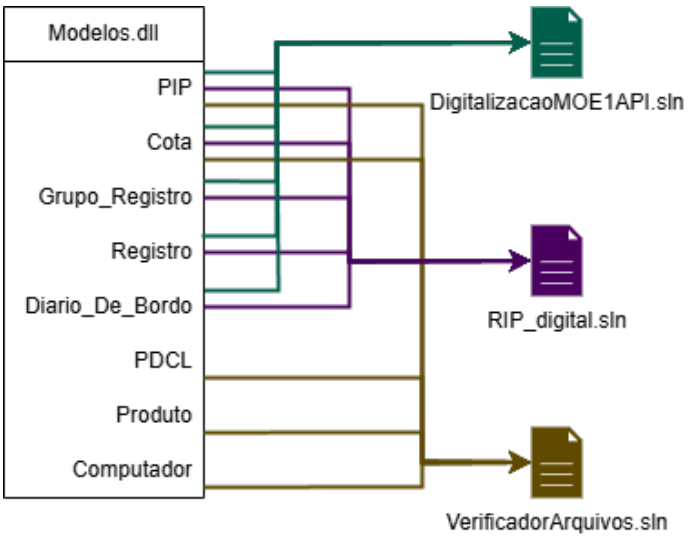
A estrutura demonstra conformidade com a 3FN (Terceira Forma Normal) na maioria das tabelas, com objetivo de reduzir redundâncias e dependências transitivas. Entretanto, a tabela `computadores` contém dependências externas que dependem da integridade de `pdcls` e `produtos`. Com isso, interpreta-se este como um ponto crítico em termos de integridade cruzada, que cria um ciclo no diagrama. A opção por manter tal característica se deve ao método pelo qual novos computadores são cadastrados: O administrador da aplicação acessa a tabela de forma direta para cadastro e exclusão de computadores. Compreende-se que, após a carga inicial de registros da tabela, não serão frequentes atualizações da mesma.

3.2 PROJETO DA BIBLIOTECA DE CLASSES `MODELOS`

Como primeiro componente da solução, o projeto de biblioteca de classes `Modelos` foi desenvolvido para centralizar as definições das classes que foram implementadas para representar entidades da base de dados, de modo a prover modularidade e reutilização no sistema. Esse projeto é compilado em uma biblioteca dinâmica (DLL), utilizada pelos outros três componentes da solução. Tal abordagem possibilita, em casos de atualizações no modelo de dados, modificar o projeto `Modelos` e recompilar a DLL, sem a necessidade

de alterar ou recompilar outros projetos dependentes. Na Figura 10, ilustra-se a utilização das classes dos modelos, por parte de cada componente.

Figura 10 – Utilização dos modelos por componente da solução.



Fonte: Autor, 2024

3.2.1 Estrutura das Classes

O arquivo principal do projeto contém as classes que representam as entidades do sistema, cada uma associada a uma tabela na base de dados. Essas classes utilizam atributos nativos do *Entity Framework Core* (EFC), como `Key`, `ForeignKey`, e `NotMapped`, para configurar propriedades e relacionamentos. Enfatiza-se que, por se tratarem de modelos de acesso a dados, nenhuma das classes possui construtores ou propriedades inicializadas. Em alguns casos, no entanto, são definidos métodos estáticos, com funcionalidades LINQ para consultas na base de dados.

A seguir, apresenta-se uma classe do projeto para expor características comuns entre elas.

3.2.1.1 Exemplo: Classe `PIP`

A classe `PIP` exemplifica a integração entre atributos, propriedades de navegação e métodos personalizados. Tal classe representa os PIPs, e inclui definições como:

```
public class PIP
{
    [Key]
    public int id_documento { get; set; }
    [ForeignKey("PDCL")]
    public string pdcl { get; set; }
```

```
    [ForeignKey("Produto")]
    public string produto { get; set; }
    ...
    [NotMapped]
    [JsonIgnore]
    public virtual ICollection<Cota>? cotas { get; set; }
    ...
```

E, como método estático, tem-se:

```
public static async Task<PIP> FindConflictingPIP(PIP newPip, DbContext
    context)
    {
        return await context.Set<PIP>()
            .Where(p => p.identificador == newPip.identificador && p.
                revisao == newPip.revisao)
            .FirstOrDefaultAsync();
```

Dentre as características apresentadas, enfatiza-se alguns aspectos comuns entre os modelos:

- Atributos e Propriedades: A utilização de atributos como `Key` (chave primária) e `ForeignKey` (chave estrangeira), que compõem a sintaxe da biblioteca `DataAnnotations` do EFC, capacita as classes para deterem metadados em seus modelos.
- Propriedades de Navegação: A coleção `cotas` representa o relacionamento de um-para-muitos com `PIP`, sem sere persistido na base, graças ao atributo `NotMapped`. Além disso, no contexto do EFC, essa prática possibilita que o código de aplicação navegue entre as entidades relacionadas sem requisitar a escrita direta de consultas SQL.
- Método Personalizado: O método `FindConflictingPIP` viabiliza a detecção de conflitos na base de dados, como duplicidade de chaves compostas. Outros métodos podem ser definidos e gerados com a DLL.

3.3 IMPLEMENTAÇÃO DA API COM A PLATAFORMA ASP.NET

O desenvolvimento da API seguiu as diretrizes da arquitetura REST, com implementação através da plataforma ASP.NET Core, que oferece uma estrutura robusta e extensível para a criação de APIs modernas. Essa plataforma é amplamente reconhecida por sua performance otimizada, suporte multiplataforma e compatibilidade com diversos padrões e protocolos do mercado, o que a torna ideal para projetos de alta complexidade e demandas de integração.

As subseções a seguir detalham aspectos técnicos da lógica desse projeto.

3.3.1 Estrutura do `Program.cs`

No projeto da API, o objeto `builder`, instanciado no arquivo `Program.cs`, desempenha a função de configurar e inicializar a aplicação ASP.NET Core. Consiste em parte do modelo de hospedagem minimalista introduzido no .NET 6, que integra as etapas de inicialização, configuração de serviços e construção do pipeline de *middleware* em uma única estrutura.

3.3.1.1 Configuração do objeto `builder`

O `builder` é instanciado com o método `WebApplication.CreateBuilder(args)`, que combina funcionalidades do host genérico e do host da Web. Com isso, facilita a leitura de configurações do ambiente e arquivos como `appsettings.json`, além de registrar serviços e middlewares essenciais. No contexto deste projeto, o `builder` é utilizado para configurar:

- **Controladores e Serialização JSON:** Os controladores, responsáveis por gerenciar as requisições HTTP, são registrados com o método `AddControllers`. Adicionalmente, a serialização JSON é configurada para desabilitar a capitalização automática dos nomes das propriedades, realizada pelo *framework*:

```
builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.PropertyNamingPolicy
            = null;
    });
```

  Essa configuração assegura compatibilidade entre modelos e padrões já estabelecidos no projeto, de modo a evitar inconsistências em integrações.

- **Contexto do Banco de Dados:** A API utiliza o *Entity Framework Core* para acessar o banco de dados. O `builder` registra o `DigitalizacaoMOE1Context` como um serviço e configura o provedor SQL Server para leitura da *string* de conexão, definida no arquivo `appsettings.json`:

```
builder.Services.AddDbContext<DigitalizacaoMOE1Context>(
   options =>
    options.UseSqlServer(builder.Configuration.
       GetConnectionString("DefaultConnection")));
```

  Essa abordagem abstrai a complexidade da interação com o banco, permitindo que o foco permaneça na lógica funcional.

- **Documentação com Swagger/OpenAPI:** O `builder` registra o Swagger, ferramenta que facilita a documentação automática dos *endpoints* da API:

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

Essa configuração fornece uma interface interativa para explorar e testar os serviços implementados, como descrito anteriormente.

3.3.1.2 Configuração do Pipeline de Requisição HTTP

Após a configuração dos serviços no objeto `builder`, através do método `builder.Build()`, o pipeline de requisição HTTP é definido pelo objeto `app`. O pipeline é responsável por gerenciar o fluxo de requisições e respostas dentro da aplicação, integrando *middlewares* para adicionar funcionalidades como segurança, roteamento e manipulação de erros. No contexto deste projeto, o pipeline de requisição HTTP é configurado como segue:

- **Ambiente de Desenvolvimento:** Durante o desenvolvimento, o pipeline inclui o uso do Swagger para facilitar a documentação e teste dos *endpoints*:

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Essa configuração garante que a interface interativa de documentação esteja disponível apenas em ambientes de desenvolvimento, prevenindo a exposição desnecessária em produção.

- **Redirecionamento HTTPS:** O `app.UseHttpsRedirection()` redireciona todas as requisições HTTP para HTTPS, a fim de garantir a segurança das comunicações entre cliente e servidor.

```
app.UseHttpsRedirection();
```

- **Autorização:** O middleware de autorização é adicionado ao pipeline para verificar e aplicar regras de acesso aos recursos da API. Ainda que este projeto não inclua autenticação complexa, essa configuração prepara o pipeline para cenários futuros:

```
app.UseAuthorization();
```

- **Mapeamento de Controladores:** Por fim, o pipeline mapeia os controladores configurados, registrando os *endpoints* definidos na aplicação:

```
app.MapControllers();
```

Essa etapa assegura que as rotas configuradas nos controladores sejam expostas corretamente e possam responder às requisições HTTP.

A configuração do pipeline no `Program.cs` viabiliza a integração dos serviços necessários de maneira eficiente e escalável, uma vez que possibilita adicionar ou remover componentes (*middlewares*) de forma dinâmica e modular, sem a necessidade de alterar a estrutura central da aplicação. Essa abordagem modular possibilita personalizar o fluxo de requisições conforme os requisitos específicos da aplicação, de modo a enaltecer a segurança e funcionalidade do projeto.

3.3.2 Estrutura do `DigitalizacaoMOE1Context.cs`

O arquivo `DigitalizacaoMOE1Context.cs` contém a classe do contexto, que representa o ambiente de execução e a configuração da aplicação. Tal classe, derivada de `DbContext`, compõe o *Entity Framework Core* e é responsável por gerenciar a interação entre a aplicação e a base de dados. A seguir, detalham-se os principais aspectos dessa implementação.

3.3.2.1 Estrutura da classe `DigitalizacaoMOE1Context`

A classe `DigitalizacaoMOE1Context` define propriedades do tipo `DbSet<T>`, que correspondem às tabelas do banco de dados. Cada `DbSet` permite a manipulação de entidades específicas, como inserção, atualização, exclusão e consultas. No exemplo, são configurados os seguintes conjuntos de entidades:

```
public DbSet<Modelos.PIP> PIPs { get; set; }
public DbSet<Modelos.Cota> Cotas { get; set; }
...
```

Essas propriedades permitem que o *Entity Framework Core* traduza as operações de alto nível em comandos SQL para interação com o banco de dados. Enfatizam-se, também, os vínculos das propriedades com as classes definidas no projeto `Modelos.sln`. Com essa abordagem, a necessidade do uso de bibliotecas ou APIs para executar comandos SQL é extinguida.

Ademais, o método `OnModelCreating` dessa classe é sobrescrito para configurar o mapeamento das entidades e suas relações. Esse método utiliza o `ModelBuilder`, classe nativa do EFC, que permite definir relacionamentos e comportamentos específicos. A seguir, apresenta-se um exemplo que ilustra a definição de um relacionamento do tipo um-para-muitos e outro do tipo um-para-um, utilizados no projeto:

- **Relacionamento entre `PIP` e `Cota`:** Um `PIP` pode conter várias `Cotas`, e cada `Cota` é associada a um `PIP`. O relacionamento é configurado como `Cascade Delete`, de modo que a exclusão de um `PIP` automaticamente remova suas `Cotas`.

```
    modelBuilder.Entity<PIP>()
        .HasMany(p => p.cotas)
```

```
        .WithOne(c => c.pip)
        .HasForeignKey(c => c.id_documento)
        .OnDelete(DeleteBehavior.Cascade);
```

- **Relacionamento entre `Diario_De_Bordo` e `Registro`:** Esse relacionamento é do tipo um-para-um, e a exclusão de um `Registro` também exclui o `Diario_De_Bordo` associado.

```
    modelBuilder.Entity<Diario_De_Bordo>()
        .HasOne(g => g.registro)
        .WithOne(r => r.diarioDeBordo)
        .HasForeignKey<Diario_De_Bordo>(r => r.id_registro)
        .OnDelete(DeleteBehavior.Cascade);
```

Essas definições asseguram integridade referencial no banco de dados, de modo a automatizar a gestão de exclusões e simplificando a lógica da aplicação.

Por fim, o construtor da classe aceita um parâmetro do tipo `DbContextOptions<T>`, que é fornecido à classe base. Essa abordagem permite que o *Entity Framework Core* configure dinamicamente a *string* de conexão e outras opções do contexto via injeção de dependência:

```
public DigitalizacaoMOE1Context(DbContextOptions<
   DigitalizacaoMOE1Context> options)
    : base(options) { }
```

Essa implementação assegura que as configurações do contexto, como a *string* de conexão (apresentada na Seção 3.3.1.1) e o provedor de banco de dados, sejam determinadas externamente, o que favorece a separação de responsabilidades e possibilita ajustes sem alterações diretas na classe.

3.3.3 Estrutura do `DBController.cs`

A classe `DBController`, contida no arquivo `DBController.cs`, é um controlador da API responsável por gerenciar as requisições relacionadas às entidades do projeto. Essa classe herda `ControllerBase`, nativa do *framework* ASP.NET, e utiliza o contexto de banco de dados `DigitalizacaoMOE1Context` para realizar operações CRUD e consultas específicas. No total, foram implementados doze métodos para realizar todas as operações da aplicação.

A seguir, apresenta-se um exemplo de método definido no controlador, que consiste em uma rota da API.

3.3.3.1 Exemplo: Método `GetPCInfo`

O método `GetPCInfo` realiza uma busca por um objeto da entidade `Computador` com base no valor do parâmetro `hostname` que, dentro da solução, é a variável de ambiente

hostname do computador do usuário:

```
[HttpGet("host")]
public async Task<ActionResult<Modelos.Computador>> GetPCInfo([FromQuery
    ] string hostname)
{
    var computador = await _context.Computadores
        .FirstOrDefaultAsync(c => c.hostname == hostname);

    if (computador == null)
    {
        return NotFound();
    }

    return computador;
}
```

Nesse bloco de código, enfatiza-se a presença de:

- Atributo `HttpGet("host")`: Define a rota do endpoint como `api/db/host`, onde `api/db` é a rota principal da API do projeto.
- Consulta na base de dados: Utiliza o método `FirstOrDefaultAsync` do *Entity Framework Core* para buscar o primeiro registro que satisfaça a condição especificada.
- Tratamento de erros: Caso nenhum registro seja encontrado, o método retorna o status HTTP 404 (`NotFound`).
- Resposta: Em caso de sucesso, o objeto `Computador` encontrado é retornado no corpo da resposta.

3.4 IMPLEMENTAÇÃO DO *WORKER SERVICE*

O projeto conta com a implementação de um *Worker Service*, uma aplicação projetada para executar tarefas de longa duração ou em segundo plano. No .NET Core, o *Worker Service* é baseado no *host genérico*, que fornece uma infraestrutura robusta para gerenciar serviços como *threads*, ciclos de execução contínuos e integração com serviços externos. Essa abordagem é amplamente utilizada para automação de processos, monitoramento ou integração com APIs.

Para essa solução, o *Worker Service* monitora um diretório no *drive* corporativo da empresa, com intuito de identificar PIPs inseridos no mesmo e, em seguida, registrá-los nas tabelas *pips* e *cotas* da base de dados. Para identificá-los, baseia-se em formatações da máscara padrão, elementos nomeados no Excel e tipo de extensão do arquivo (*.xlsx*).

3.4.1 Funcionamento do Worker Service

O *Worker Service* é iniciado junto com o sistema, sendo gerenciado pelo host do .NET Core. Sua configuração é centralizada no arquivo `Program.cs`, onde são definidos os serviços a serem executados e os recursos adicionais necessários, como configurações e injeções de dependência.

No contexto deste projeto, o serviço principal, denominado `AtualizadorPIPExcel`, realiza a verificação e o processamento de arquivos, interagindo diretamente com o sistema de arquivos para garantir a atualização de dados relacionados ao sistema. O *Worker Service* é configurado para rodar continuamente, e emprega abstração de um *host* para gerenciar seu ciclo de vida.

3.4.2 Configuração do Arquivo `Program.cs`

A seguir, apresenta-se a lógica do arquivo `Program.cs`, que define as configurações iniciais do *Worker Service*. Esse arquivo utiliza o modelo minimalista do .NET, introduzido a partir do .NET 6, para integrar serviços e configurações.

```
var builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddJsonFile("appsettings.json", optional: false,
    reloadOnChange: true);

builder.Services.Configure<Settings>(builder.Configuration);

builder.Services.AddHostedService<AtualizadorPIPExcel>();

var host = builder.Build();

await host.RunAsync();
```

3.4.2.1 Aspectos Técnicos do Código

O código acima apresenta as seguintes características principais:

- **Host Genérico:** O `Host.CreateApplicationBuilder` cria o host genérico. Essa abordagem é usada para desacoplar a lógica da aplicação dos detalhes de infraestrutura e gerenciamento do ciclo de vida do *Worker Service*. Proporciona, assim, flexibilidade e extensibilidade.
- **Configuração via `appsettings.json`:** O método `AddJsonFile` carrega as configurações da aplicação, uma vez que projetos dessa categoria não possuem funcionalidade nativa para uso de arquivos de configuração. Deste modo, capacita o serviço a acessar informações como diretórios e parâmetros definidos pelo usuário.

- **Injeção de Dependência:** A linha que contém `builder.Services.Configure` utiliza o padrão *Options* para mapear a configuração da aplicação em uma classe auxiliar definida para esse propósito (`Settings`). E, para registrar o serviço como uma tarefa em segundo plano, sem comprometer a interface da máquina que o hospeda, utiliza-se o método `builder.Services.AddHostedService`.
- **Execução Contínua:** O método `RunAsync()` inicia o *Worker Service* e mantém sua execução até que seja explicitamente encerrado.

3.4.3 Resumo da lógica em `DadosPIP.cs`

O arquivo `DadosPIP` desempenha uma função primordial na leitura e processamento de dados provenientes das planilhas Excel, os PIPs, como descrito na seção 3.4. A classe centraliza a extração e validação de informações relacionadas ao cabeçalho e às cotas, mapeia os valores para as respectivas propriedades de modelos definidos, sendo estes `PIP` e `Cota`. Por meio do uso de bibliotecas como `EPPlus`, a classe interpreta faixas nomeadas e tabelas na planilha, garantindo que os dados sejam estruturados de forma adequada para o consumo posterior. O detalhamento sobre o funcionamento da extração de dados de PIPs revelaria detalhes da estrutura de um PIP que, como acordado com a empresa, não será realizado.

Uma vez que, em ambientes produtivos, haverão ocorrências de postagem de PIPs fora do padrão de leitura e/ou erros na comunicação com a API que realiza o POST dos novos documentos, o arquivo *falhas_salvamento_PIP.log* é gerado para manter histórico de erros dessas operações. A estruturação desse arquivo é feita através de uma lista JSON que registra a data/hora, código de *status* e *payload* da tentativa de cadastro. Abaixo, apresenta-se um registro gerado durante desenvolvimento:


```
...
{
  "Data": "21/10/2024 20:20:19",
  "CodigoDeStatus": 409,
  "Cabecalho": {
    "id_cota": 0,
    "id_documento": 1,
    "identificador": "22",
    "cota_critica": "",
    "descricao": "Rugosidade da face",
    "metodologia": "Variavel",
    "posto": "Bancada",
    "frequencia": "1 - 6",
    "sistema_de_medicao": "Rugosimetro",
    "resolucao": "0,025",
```

```
    "registro": "PR; TT; FL"
  }
},
...
```

3.5 APLICAÇÃO EM WINDOWS FORMS

Como componente da interface gráfica com usuário, a aplicação implementada como *Windows Forms* utiliza o modelo de programação orientada a eventos para gerenciar interações com o usuário e operações relacionadas ao processamento de dados. O *software* é projetado para possibilitar acesso aos RIPs vinculados ao computador do usuário que, através da obtenção de informações de seu respectivo PIP mediante à interações com a API, apresenta uma interface dedicada ao preenchimento dos dados de registro e das cotas, efetivamente.

3.5.1 Arquitetura Geral

A estrutura do programa é composta por componentes principais que garantem a integração com a API, o processamento de dados e a interação com o usuário:

- Integração com a API: O programa consome serviços RESTful para buscar informações de PIPs e cotas, assim como para registrar dados processados, quando as inspeções são submetidas. A classe `ApiClient.cs` gerencia todas as operações de requisição HTTP.
- Validação e Preenchimento de Dados: A aplicação valida dinamicamente as entradas do usuário para evitar erros e garantir consistência antes do envio dos dados à API.
- Interface Dinâmica: Controles como `DataGridView`, `ComboBox` e `TextBox` são utilizados para exibir, editar e gerenciar dados de forma interativa.

3.5.2 Funcionalidades Técnicas

A seguir, destacam-se as principais funcionalidades técnicas da aplicação e realizam-se menções aos respectivos controles da interface principal, apresentada na Figura 11.

3.5.2.1 Carregamento e Inicialização

Ao inicializar, a aplicação realiza requisições à API para buscar informações relacionadas ao computador. Utiliza-se a propriedade `Environment.MachineName`, contida na classe `Environment` do .NET que retorna o nome do computador (`host`) onde o programa

Figura 11 – Interface principal do RIP digital.

Fonte: Autor, 2024

está sendo executado. Essa propriedade representa o nome configurado no sistema operacional para identificar a máquina na rede ou localmente. Essas informações são apresentadas no canto inferior esquerdo da aplicação, para conhecimento do usuário.

```
protected override async void OnLoad(EventArgs e) {
...
try
{
    Modelos.Computador host = await apiClient.GetAsync<Computador>($"
       host?hostname={Environment.MachineName}");
```

```
    Hostname = host.hostname;
    PDCL = host.pdcl;
    Produto = host.produto;
    ...
```

Em seguida, de forma análoga, realiza-se uma consulta para obter os PIPs disponíveis para tal computador. Essa informação é obtida em forma de uma lista de objetos `PIP`, e cada um deles é disponibilizado para selação na `ComboBox` denominada "Tipo de Produto".

```
    ...
    pipsDisponiveis = await apiClient.GetPIPsByComputadorAsync(PDCL,
        Produto);

    cbDescricaoItem.DataSource = pipsDisponiveis;
    cbDescricaoItem.DisplayMember = "descricao_item";
    cbDescricaoItem.SelectedIndex = -1;
    ...
```

Essa etapa inicial assegura que os dados exibidos na interface estejam sincronizados com o ambiente de produção.

3.5.2.2 Validação Dinâmica de Entradas

O método `atribuirChecagemTbsECbs()`, executado após etapas da inicialização, associa eventos aos controles do cabeçalho da aplicação, para validar os dados inseridos pelo usuário. Para localizar os controles referentes aos dados de preenchimento do registro, inclui-se uma pesquisa pelas propriedades da classe `Grupo_Registro`, contida no modelo dinâmico:

```
public void atribuirChecagemTbsECbs() {
    foreach (PropertyInfo property in typeof(Grupo_Registro).
        GetProperties()) {
        var matchingControl = this.Controls.Find("tb" + property.Name,
            true).FirstOrDefault() ?? this.Controls.Find("cb" + property.
            Name, true).FirstOrDefault();

        if (matchingControl is TextBox textBox)
        {
            textBox.TextChanged += checagemPreenchimentoCabecalho;
        }
        else if (matchingControl is ComboBox comboBox)
        {
            comboBox.TextChanged += checagemPreenchimentoCabecalho;
            comboBox.SelectedIndexChanged +=
                checagemPreenchimentoCabecalho;
        }
```

```
        else
        {
            ...
            Application.Exit();
        }
    }
    btEnviar.Enabled = true;
}
```

Essa abordagem reduz a possibilidade de erro humano, uma vez que desabilita o botão de envio e previne o envio de dados incompletos e/ou fora do padrão.

3.5.2.3 Exibição e Edição de Cotas

O `DataGridView` denominado `dgvCotas` é a tabela apresentada no centro da interface. É a ferramenta aplicada para exibir as informações relevantes das `cotas` do PIP selecionado e detém uma coluna para inserção dos valores relacionados aos registros.

```
private void PopulateDataGridView(List<Cota> cotasPip) {
    ...
    foreach (var cota in cotasPip)
    {
        dgvCotas.Rows.Add(cota.identificador, cota.cota_critica, cota.
           descricao, "");
        dgvCotas.Rows[idLinha].Tag = cota;

    }
...
}
```

Para definir uma rastreabilidade quanto à `cota` contida em cada linha, utiliza-se a propriedade `Tag`, nativa de projetos Windows Forms, para armazenar o objeto correspondente ao índice da mesma. Essa implementação é útil posteriormente, quando são realizados POSTs dos registros para a API.

3.5.2.4 Envio de Registros de RIPs

O método `postCotas()`, vinculado ao evento de clique no botão "Enviar", é responsável por realizar o envio da inspeção para a API, ou seja, cria um registro na tabela `grupos_registro`, e um em `registros` para cada cota preenchida. O processo é dividido em duas etapas: o envio do grupo de registro e, posteriormente, o envio de cada registro individual associado às cotas.

Na primeira etapa, o objeto `Grupo_Registro` é instanciado e preenchido com informações extraídas dinamicamente dos controles da interface, como caixas de texto e seletores (`TextBox` e `ComboBox`), detalhadas na Seção 3.5.2.2. Após a configuração, o objeto é enviado à API utilizando o método `PostAsync`:

```
public async Task<Dictionary<Cota,int>> postCotas()
{
    ...
    Grupo_Registro grupoRegistro = new Grupo_Registro();
    grupoRegistro.id_documento = pip.id_documento;

    foreach (var (propriedade, control) in camposGrupoRegistro) {
        string conteudoControl = control.Text;
        if (propriedade.PropertyType == typeof(int) && int.TryParse(
            conteudoControl, out int intValue)) {
            propriedade.SetValue(grupoRegistro, intValue);
        } else if (propriedade.PropertyType == typeof(string)) {
            propriedade.SetValue(grupoRegistro, conteudoControl);
        }
    }
    grupoRegistro.data_hora = DateTime.Now;

    Grupo_Registro postedGrupoRegistro = await apiClient.PostAsync("
        grupo_registro", grupoRegistro);
    ...
```

Após sucesso no envio do `Grupo_Registro`, o método segue para a etapa de envio dos registros. Cada linha do `DataGridView`, que representa uma cota, é processada para criar um objeto `Registro`, o qual contém os dados necessários para a API. O `id_grupo_registro`, obtido no primeiro POST, é reutilizado para garantir a associação dos registros ao grupo previamente criado.

```
foreach (DataGridViewRow linha in dgvCotas.Rows) {
    var valValor = linha.Cells[colValor].Value;

    if (!linha.IsNewRow && valValor != null && !string.
        IsNullOrWhiteSpace(valValor.ToString())) {
        var linhaObj = (Cota)linha.Tag;

        Registro registro = new Registro() {
            id_grupo_registro = postedGrupoRegistro.id_grupo_registro,
            id_cota = linhaObj.id_cota,
            valor = valValor.ToString()
        };

        Registro postedRegistro = await apiClient.PostAsync("registro",
            registro);
        dictIdRegistro[linhaObj] = postedRegistro.id_registro;
    }
}
```

Observa-se que, como mencionado anteriormente, a propriedade `Tag` é aplicada

para relacionar a chave estrangeira `id_cota` ao registro na tabela `registros`. Além disso, um dicionário é incrementado para todo registro enviado. Essa operação é relevante para o envio de Diário de Bordo, detalhado na 3.5.3.3.

3.5.2.5 Tratamento de Erros e Reversão de Operações

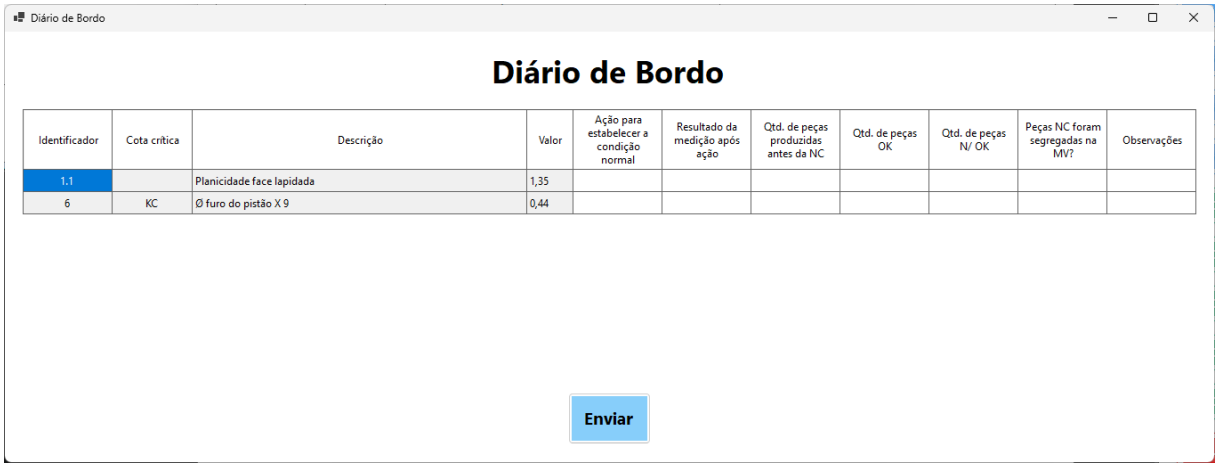
Caso ocorra um erro durante o envio de um registro no método `postCotas()`, o grupo de registro e seus registros associados são deletados para evitar inconsistências na base de dados. Essa reversão é implementada por meio de uma requisição `DELETE` à API:

```
try {
    Registro postedRegistro = await apiClient.PostAsync("registro",
       registro);
    dictIdRegistro[linhaObj] = postedRegistro.id_registro;
} catch (Exception ex) {
    await apiClient.DeleteAsync(postedGrupoRegistro);
    throw new Exception(ex);
}
```

3.5.3 Formulário de Diário de Bordo

O formulário denominado `DiárioDeBordo` é aberto quando a caixa "Não Conformidade" está selecionada na interface principal e o botão "Enviar" é pressionado. Esse formulário foi projetado para registrar dados adicionais necessários para medições não conformes, de modo a complementar as inspeções com registros adicionais na tabela `diarios_de_bordo`.

Figura 12 – Interface secundária do RIP digital (Diário de Bordo).



Fonte: Autor, 2024

A interface deste formulário, representada pela tabela `dgvDiarioDeBordo`, é construída com base nas linhas selecionadas no `DataGridView` principal (`dgvCotas`). Cada

linha contém as informações originais da cota e adiciona colunas específicas para os campos requeridos no registro de diário de bordo.

3.5.3.1 Configuração Inicial

Na inicialização do formulário, realiza-se uma cópia da estrutura de colunas e valores do `dgvCotas` para o `dgvDiarioDeBordo`.

```
public DiarioDeBordo(Main parentForm, DataGridView dgvCotas)
{
    _parentForm = parentForm;

    InitializeComponent();
    ...
    foreach (DataGridViewColumn column in dgvCotas.Columns)
    {
        DataGridViewColumn clonedColumn = (DataGridViewColumn)column.
            Clone();
        clonedColumn.ReadOnly = true;
        clonedColumn.DefaultCellStyle.BackColor = Color.FromArgb(240,
            240, 240);
        dgvDiarioDeBordo.Columns.Add(clonedColumn);
    }
    ...
```

Em seguida, novas colunas são adicionadas para incluir campos requeridos no modelo `Diario_De_Bordo`, como *Ação para estabelecer a condição normal*, *Resultado da medição após ação* e *Qtd. de peças N/OK*, entre outros. As informações originais são mantidas como somente leitura, de modo a evitar edições indevidas.

```
    ...
    int indexHeader = 0;
    foreach (PropertyInfo propriedade in typeof(Diario_De_Bordo).
        GetProperties().OrderBy(p => p.MetadataToken))
    {
        if (!propriedade.GetCustomAttributes().Any())
        {
            colsDiarioDeBordo.Add("col" + propriedade.Name,
                dgvDiarioDeBordo.Columns.Add(new
                DataGridViewTextBoxColumn { HeaderText =
                lstHeadersDiarioDeBordo[indexHeader] }));
            indexHeader++;
        }
    }
    ...
```

3.5.3.2 Validação de Campos

Antes do envio, o método `validarPreenchimentoDiario` verifica se todas as células referentes às propriedades do modelo `Diario_De_Bordo` estão preenchidas. Caso algum campo esteja vazio, uma mensagem de erro é exibida e o processo é interrompido.

```
private bool validarPreenchimentoDiario()
{
    ...
        foreach (PropertyInfo propriedade in typeof(Diario_De_Bordo).
           GetProperties())
        {
            var valorPropriedade = linha.Cells[colsDiarioDeBordo["col" +
                propriedade.Name]]?.Value?.ToString();
            if (string.IsNullOrWhiteSpace(valorPropriedade))
            {
                MessageBox.Show($"Favor, preencher {propriedade.Name}.")
                    ;
                return false;
            }
        }
    ...
    return true;
}
```

3.5.3.3 Envio de Dados

O botão "Enviar Diário de Bordo" aciona o método `btEnviarDiario_Click`, que valida o preenchimento de todas as colunas adicionais antes de enviar os registros para a API. Cada linha da tabela é associada a um objeto `Diario_De_Bordo`, preenchido dinamicamente com base nas propriedades do modelo. Em seguida, o registro é enviado ao *endpoint* correspondente da API.

```
private async void btEnviarDiario_Click(object sender, EventArgs e)
{
    if (!validarPreenchimentoDiario()) return;

    foreach (DataGridViewRow linha in dgvDiarioDeBordo.Rows)
    {
        int idRegistro = dictIdRegistro[(Cota)linha.Tag];

        Diario_De_Bordo diarioDeBordo = new Diario_De_Bordo();
        diarioDeBordo.id_registro = idRegistro;

        foreach (PropertyInfo propriedade in typeof(Diario_De_Bordo).
           GetProperties())
        {
```

```
            var valorPropriedade = linha.Cells[colsDiarioDeBordo["col" +
                propriedade.Name]]?.Value?.ToString();
            if (propriedade.PropertyType == typeof(int) && int.TryParse(
                valorPropriedade, out int intValue))
            {
                propriedade.SetValue(diarioDeBordo, intValue);
            }
            else if { ... }
            ...
        }

        await apiClient.PostAsync("diario_de_bordo", diarioDeBordo);
    }
}
```

4 RESULTADOS

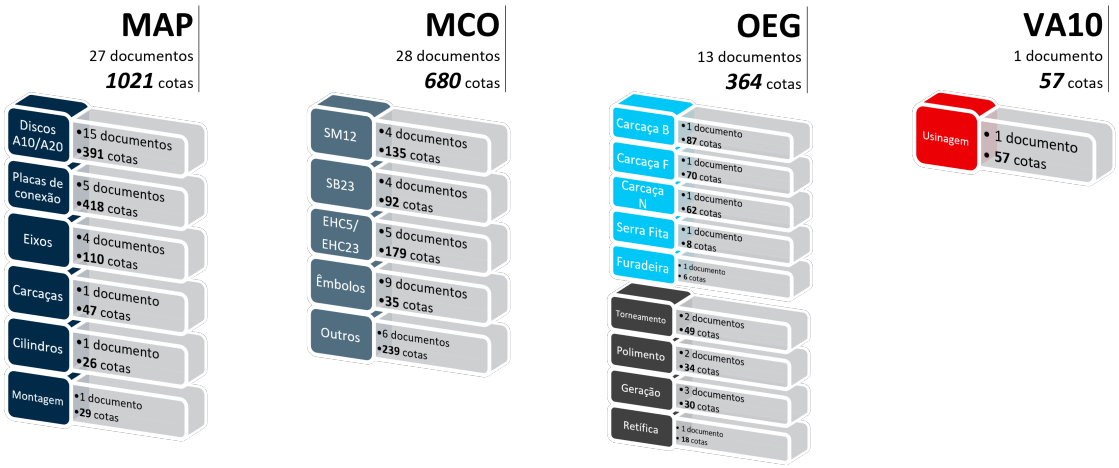
Após a modelagem da base de dados relacional, desenvolvimento e integração dos componentes da solução, e validação do sistema em ambiente local, faz-se pertinente avaliar os resultados com fundamento nas adequações das funcionalidades da primeira instância e validação das características inéditas. As seções a seguir são dedicadas para tais análises.

4.1 FUNCIONALIDADES DA PRIMEIRA INSTÂNCIA

Como explicitado na Seção 1.2, as funcionalidades básicas da primeira instância do projeto compuseram o fundamento inicial para transição de uma solução via Excel para um *software* dedicado ao envio dos RIPs. Neste contexto, dentre as seis características presentes na primeira versão, quatro foram implementadas de forma equivalente ou aprimorada:

- **Preenchimento automático e travamento de data/hora da submissão:** A data/hora de submissão de registros, presente na primeira instância como uma célula bloqueada com valor de data/hora do início do preenchimento, está contida em um atributo próprio na tabela `grupos_registro`, denominada `data_hora`. Optar por essa abordagem agrega na responsividade da aplicação, uma vez que subrotinas do tipo *Worksheet_Change* eram utilizadas para verificar em tempo real se o usuário tinha iniciado o preenchimento de uma inspeção.
- **Registro dinâmico e diário de bordo:** Como descrito no desenvolvimento, uma interface secundária dedicada a essa funcionalidade foi concebida. Ademais, a divisão de tabelas em `registros` e `diarios_de_bordo` viabiliza a decomposição de registros com não conformidades em ferramentas para análise de dados, recurso que não era compatível com a outra versão, uma vez que "registros compostos" eram formatados de modo precarizado em tabelas *.xlsx*.
- **Centralização e criação de tabelas dedicadas para cada revisão do RIP:** Sendo o aprimoramento mais notável do projeto pois, além de enaltecer o desempenho, integridade e escalabilidade dos registros por substituir as tabelas *.xlsx* por uma base de dados relacional, sucede-se o ganho significativo atrelado à obsolescência de documentos não mais necessários na plataforma de gestão da empresa. Esse resultado é ilustrado na Figura 13, que apresenta um sumário em relação à aplicação dos RIPs na planta da Bosch Rexroth em Pomerode. Observa-se que 69 documentos serão obsoletados, distribuídos entre quatro áreas (MAP, MCO, OEG e VA10). Isso se deve ao fato da solução desenvolvida não requerer o cadastro de um documento exclusivo para um RIP, uma vez que a mesma utiliza os dados de um PIP para gerar uma interface de preenchimento dos registros.

Figura 13 – Impacto da solução na Bosch Rexroth de Pomerode.



Fonte: Autor, 2024

- **Proteção por senha e modo somente leitura:** Tal funcionalidade fazia-se necessária no contexto em que arquivos do Excel estão sujeitos à edição caso o usuário do sistema possua acesso de gravação no diretório em questão. Uma vez que a aplicação trabalha com controles pré-programados na interface gráfica, uma tratativa de proteção não se fez necessária.

Enfatiza-se que, no entanto, no decorrer do desenvolvimento do Trabalho, duas funcionalidades foram suprimidas:

- **Temporizador parametrizável para acompanhamento de processo:** O desenvolvimento do temporizador, que se tratava de uma solução para auxiliar no controle da frequência de inspeções dentro da fábrica, foi abortado devido a implantação de um novo sistema de gerenciamento de produção (ou MES, *Manufacturing Execution System*) durante o segundo semestre de 2024, pelo departamento da Engenharia de Qualidade. Fazê-lo tornaria a função redundante, o que não integraria ao sistema homologado da planta.
- **Notificação automática de falta de registros:** Após o *design* do esquema da base de dados, verificou-se que, devido à capacidade do sistema de operar em conjunto com ferramentas de análise de dados baseados na Web, é preferível optar por uma solução dedicada à realização de fluxos automatizados com fundamento na leitura de dados da base, como o desenvolvimento de um fluxos de e-mail através do Microsoft Power Automate, ou um *chatbot* de notificações via Microsoft Power Agents.

4.2 FUNCIONALIDADES INÉDITAS

A população da base de dados foi realizada com três PIPs escolhidos arbitrariamente, os quais detêm metadados conforme Figura 14 (exceto pelos nomes reais dos autores, os quais foram omitidos).

Figura 14 – Metadados dos PIPs considerados.

id_documento	identificador	pdcl	produto	revisao	descricao_item	maquinas	codigo_item	desenho	data	elaborador	documentos_relacionados
1046	PIP-AK-CIL-002	MAP	Cilindro	5	Cilindros A10	["Lapidadora e inspe\u00E7\u00E3o"]	Vide codigo do item	Vide desenho do item	14/07/2022	Pedro Souza	["RIP-AK-CIL-002","CEP-AK-CIL-004","CEP-AK-CIL-001","CEP-AK-CIL-002"]
1048	PIP-AK-BOM-001	MAP	Bomba	4	Diversos	["Linha de montagem"]	Diversos	Diversos	20/05/2024	Matheus Oliveira	RIP-AK-BOM-001
1049	PIP-AK-CIL-001	MAP	Cilindro	5	Cilindro A10	["2348","2349","6201","6202"]	Vide código	Vide código	24/08/2023	Gabriela Mendes	["RIP-AK-CIL-001","CC-AK-CIL-003"]

Fonte: Autor, 2024

Enfatiza-se que o tempo de processamento médio para registro de novas inspeções na base de dados situou-se entre 40–50 ms, enquanto o tempo para envios da implementação via Excel constata-se entre 1,2–1,4 s. Ambos resultados foram apurados em tabelas vazias, amostradas ao menos 10 vezes, em ambiente local. O ganho de tempo por envio foi estimado em aproximadamente 29 vezes em relação à implementação original. Estes resultados devem-se à combinação tecnologias empregadas para desenvolvimento do sistema dedicado, sendo a API RESTful em conjunto com a base de dados SQL uma alternativa mais eficiente em comparação ao uso de macros e bases de dados Excel. Ademais, estima-se que o processo de abertura e digitação de valores no programa é de aproximadamente uma vez e meia mais rápido que a operação realizada pela primeira instância do programa, via Excel. Ao considerar essa informação, dado que 69 RIPs são utilizados na fábrica, com 200 dias de produção inspecionada anualmente, um ganho teórico de cerca de 194 horas é obtido.

Por fim, a implementação do sistema proporcionou ganhos qualitativos significativos ao introduzir uma abordagem moderna e robusta para o gerenciamento e processamento de dados. Ao utilizar APIs RESTful integradas à base de dados relacional, o sistema garantiu maior integridade e consistência dos dados, além de otimizar a organização e o acesso às informações. A adoção de ferramentas como o *Entity Framework Core* e o LINQ promoveram a abstração das operações da base, de forma a reduzir a complexidade do código e minimizar erros. Essa estrutura modular, escalável e amplamente documentada simplifica a manutenção, amplia a capacidade de expansão do sistema e assegura um desempenho confiável, mesmo em cenários de grande volume de dados e alta demanda.

5 CONCLUSÃO

O presente trabalho apresentou o desenvolvimento de uma solução integrada para a digitalização de processos industriais, com foco no gerenciamento de Registros de Inspeção de Processo (RIPs) e Planos de Inspeção de Processo (PIPs). A aplicação projetada e implementada, utilizando tecnologias como *Windows Forms*, *Entity Framework Core* e APIs RESTful, foi concebida para atender às necessidades do setor produtivo, de modo a promover melhorias significativas em eficiência, rastreabilidade e conformidade.

No aspecto funcional, a aplicação *Windows Forms* desempenhou um papel primordial como interface gráfica com o usuário, facilitando o acesso e o preenchimento de informações relacionadas aos RIPs. Com funcionalidades como validação dinâmica de entradas, exibição e edição de cotas por meio de `DataGridView`, e envio de registros para a API, o sistema foi projetado para minimizar erros humanos e otimizar fluxos de trabalho. A implementação de formulários adicionais, como o `DiarioDeBordo`, permitiu o registro detalhado de medições não conformes, atendendo a requisitos específicos de controle de qualidade.

Outro ponto de destaque foi a aplicação de boas práticas de programação e o uso de recursos como injeção de dependência, abstração de eventos e tratamento de erros. Esses elementos garantiram a escalabilidade e a manutenibilidade do sistema, permitindo sua adaptação futura a novos requisitos e integrações.

Enfatiza-se que o trabalho alcançou os objetivos propostos, demonstrando a viabilidade de substituir processos manuais e baseados em papel por uma solução digital eficiente e integrada. O sistema não apenas otimizou os fluxos de trabalho existentes, como também abriu possibilidades para expansões futuras, como a integração com sistemas de inteligência artificial para análise de dados ou a migração para uma arquitetura baseada em nuvem. Ademais, o projeto contribui significativamente para a modernização de processos industriais da empresa, alinhando-se às tendências da Indústria 4.0 e destacando-se como um exemplo prático de como a tecnologia pode ser aplicada para melhorar a produtividade e a qualidade nos processos produtivos.

Por fim, estima-se que a adesão da solução por parte da empresa deve ocorrer no decorrer do segundo semestre de 2025, de modo a possibilitar um período prévio de adequações dos componentes em ambiente oficial da intranet da empresa, como criação da base no servidor do departamento de informática, implementação de acesso via *Active Directory*, empregado na rede corporativa e validação da escalabilidade do sistema, dada a presença de muitos PIPs.

REFERÊNCIAS

CODD, Edgar F. Recent Investigations in Relational Data Base Systems. *In*: PROCEEDINGS of the ACM Pacific Conference. San Francisco, CA: ACM, 1974.

CORONEL, Carlos; MORRIS, Steven. **Database Systems: Design, Implementation, and Management**. 13th. Boston, MA: Cengage Learning, 2019.

GULATI, Vaibhav. **Introduction to Relational Model in DBMS**. 2024. Disponível em: https://www.scaler.com/topics/introduction-to-relational-model. Acesso em: 18 dez. 2024.

HUDLI, Shrinidhi R.; HUDLI, Raghu V. A Verification Strategy for Dependency Injection. **Lecture Notes on Software Engineering**, v. 1, n. 1, 2013.

KUMAR, Kunal; AZAD, S. K. **Database Normalization Design Pattern**. 2017. Disponível em: https://ieeexplore.ieee.org/abstract/document/8251067. Acesso em: 17 out. 2024.

LIMA, Faíque Ribeiro; GOMES, Rogério. **Conceitos e tecnologias da Indústria 4.0**: uma análise bibliométrica. 2020. Disponível em: https://www.scielo.br/j/rbi/a/x6jdz4t869KnNFWRdgqVyws/?lang=pt. Acesso em: 2 out. 2024.

LINSINGEN, Irlan von. **Fundamentos de Sistemas Hidráulicos**. 5. ed. Florianópolis: Editora UFSC, 2016. P. 398.

MICROSOFT LEARN. **Entity Framework Core**. 2024. Disponível em: https://learn.microsoft.com/pt-br/ef/core/. Acesso em: 16 out. 2024.

MICROSOFT LEARN. **Implementar a camada de persistência de infraestrutura com o Entity Framework Core**. 2024. Disponível em: https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implementation-entity-framework-core. Acesso em: 20 nov. 2024.

MICROSOFT LEARN. **LINQ (Consulta Integrada à Linguagem)**. 2024. Disponível em: https://learn.microsoft.com/pt-br/dotnet/csharp/linq/. Acesso em: 24 out. 2024.

MICROSOFT LEARN. **Middleware do ASP.NET Core**. 2024. Disponível em: https://learn.microsoft.com/pt-br/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0. Acesso em: 21 out. 2024.

MICROSOFT LEARN. **Serviços de trabalho no .NET**. 2024. Disponível em: https://learn.microsoft.com/pt-br/dotnet/core/extensions/workers. Acesso em: 12 out. 2024.

MICROSOFT LEARN. **Tutorial: Create a web API with ASP.NET Core**. 2024. Disponível em: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-9.0&tabs=visual-studio. Acesso em: 17 nov. 2024.

ORDONEZ, Carlos; GARCÍA-GARCÍA, Javier. Referential integrity quality metrics. **Decision Support Systems**, Elsevier, v. 44, p. 495–508, 2008.

QURESHI, M. Rizwan Jameel; SABIR, Fatima. A comparison of model view controller and model view presenter. **Science International-Lahore**, v. 25, p. 7–9, 2014.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services**. Sebastopol, CA: O'Reilly Media, 2007.

SENSIO. **Manufatura inteligente:** o que é e como aplicar. 2024. Disponível em: https://www.sensio.com.br/blog/manufatura-inteligente-o-que-e-e-como-aplicar#:~:text=Aplicar%20a%20manufatura%20inteligente%20envolve,e%20do%20desenvolvimento%20de%20compet%C3%AAncias.. Acesso em: 20 out. 2024.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Database System Concepts**. 7th. New York, NY: McGraw-Hill Education, 2019.

SWAGGER. **Swagger - API Documentation Tools**. [*S.l.*: *s.n.*]. Disponível em: https://swagger.io/. Acesso em: 26 nov. 2024.

ZAKLOUTA, Hadi. **Cost of Quality Tradeoffs in Manufacturing Process and Inspection Strategy Selection**. 2011. Master's thesis – Massachusetts Institute of Technology, Cambridge, MA.