Data lake para agregação de dados de produção e ferramentas de visualização na indústria de estampagem

Leonardo Leite Meira dos Santos - 54363

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação.

Trabalho orientado por:

Prof. Paulo Alves

Prof. Kecia Marques

Esta dissertação não inclui as críticas e sugestões feitas pelo Júri.

Bragança

2022-2023

Data lake para agregação de dados de produção e ferramentas de visualização na indústria de estampagem

Leonardo Leite Meira dos Santos - 54363

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação.

Trabalho orientado por:

Prof. Paulo Alves

Prof. Kecia Marques

Esta dissertação não inclui as críticas e sugestões feitas pelo Júri.

Bragança

2022-2023

Dedicatória

(Facultativo) Dedico este trabalho a ...

Agradecimentos

(Facultativo) Agradeço a ...

Resumo

O resumo (no máximo com 250 palavras), permite a avaliação do interesse de um documento e facilita a sua identificação na pesquisa bibliográfica em bases de dados onde o documento se encontre referenciado.

É recomendável que o resumo aborde, de forma sumária:

- Objetivos principais e tema ou motivações para o trabalho;
- Metodologia usada (quando necessário para a compreensão do relatório);
- Resultados, analisados de um ponto de vista global;
- Conclusões e consequências dos resultados, e ligação aos objetivos do trabalho.

Como este modelo de relatório se dirige a trabalhos cujo foco incide, maioritariamente, no desenvolvimento de software, algumas destas componentes podem ser menos enfatizadas, e acrescentada informação sobre análise, projeto e implementação do trabalho.

O resumo não deve conter referências bibliográficas.

Palavras-chave: termos (no máximo 4), que descrevem o trabalho.

Abstract

Direct translation (maximum of 250 words) to English of the section “Resumo”.

Keywords: direct translation of “Palavras-chave”

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Enquadramento

No cenário industrial contemporâneo, a busca constante por eficiência e inovação tornou-se um pilar essencial para a competitividade e sustentabilidade financeira das empresas. À medida que a tecnologia avança, a produção industrial sofre uma evolução para se manter competitiva perante ao mercado. Nesse contexto, o monitoramento e a otimização das máquinas em linhas de produção tornam-se cruciais para garantir um funcionamento eficaz e prevenir potenciais paralisações ou falhas operacionais.

No entanto, a tradição das práticas industriais muitas vezes é caracterizada por inspeções manuais e sistemas de monitoramento desatualizados, que não conseguem fornecer informações em tempo real ou análises aprofundadas sobre o desempenho das máquinas. Esse descompasso tecnológico pode resultar em perdas significativas, em termos de produção, de recursos financeiros e manutenção de máquinas.

Além disso, com a crescente integração de sistemas de IoT e a proliferação de sensores avançados, existe uma quantidade imensa de dados sendo gerada continuamente. No entanto, sem a infraestrutura adequada para coletar, armazenar e analisar esses dados, as empresas podem se encontrar sobrecarregadas, incapazes de extrair insights significativos que poderiam informar decisões estratégicas e operacionais.

Com esse contexto de mercado, a empresa Catraport buscou a construção de projetos que realizem a sensorização de suas maquinas de estampagem, armazenamento e tratamento dos dados e visualização dessas informações, de forma a se tornar mais competitiva, eficiente e lucrativa.

1.2 Objetivos

Dado o enquadramento anterior, a necessidade identificada é de desenvolver um sistema robusto que possa receber dados de sensores que coletam dados das máquinas em tempo real, armazená-los de maneira eficiente em um data lake e apresentá-los através de um dashboard, transformando a maneira como as empresas monitoram e otimizam suas linhas de produção, garantindo não apenas eficiência, mas também uma abordagem proativa à manutenção e gestão industrial. Esse sistema não apenas forneceria informações em tempo real sobre o status e o desempenho das máquinas, mas também permitiria análises históricas, ajudando gestores e técnicos a identificar tendências e falhas, além de otimizar a produção, minimizando perdas na produção e otimizando os ganhos financeiros.

1.3 Estrutura do Documento

Esta dissertação de mestrado está organizada começando pela Introdução, onde o problema e o escopo são apresentados. Esta seção contextualiza a necessidade de modernização industrial, destacando a importância do monitoramento e otimização de máquinas na indústria. A relevância do estudo é então justificada, com foco na crescente demanda por análise de dados no mercado.

Segue-se com a Revisão Literária, que apresenta uma análise sobre trabalhos e conceitos relacionados ao projeto. Esta seção engloba desde trabalhos semelhantes já realizados, métodos de análise de dados, até discussões sobre visualização de dados e importância da gestão de dados no mercado atual.

A Metodologia discute a escolha de tecnologias específicas usadas no projeto, detalhando o método de armazenamento de dados, e o processo adotado para o desenvolvimento do software. Também são descritas as estratégias de gestão das atividades do projeto e os desafios enfrentados durante a coleta de dados.

No capítulo sobre a Arquitetura do Sistema, é discutido sobre as especificidades técnicas do sistema proposto. Desde o diagrama geral do sistema até a implementação de princípios de codificação, como o SOLID, esta seção visa elucidar o design e a funcionalidade do software criado.

A seção de Implementação aprofunda-se nos aspectos técnicos do sistema. Aqui, cada componente do banco de dados, da API e do módulo de processamento de dados é detalhadamente descrito. Também se discute a reutilização potencial do sistema em outros contextos, e o que seriá necessário para isso.

No que diz respeito às Características do Sistema do ponto de vista funcional, a dissertação tem uma seção que foca em mostrar a aplicação prática do software, utilizando capturas de tela e diagramas para demostrar as funcionalidades.

Em Resultados e Avaliação, são apresentados os resultados obtidos durante o projeto, destacando as principais realizações e benefícios observados na implementação do sistema proposto. A Conclusão e Trabalhos Futuros resume os pontos chave do projeto, identifica as limitações do sistema atual e sugere possíveis direções para continuidade e implementações futuras.

Capítulo 2

Revisão literaria

In this chapter it is expected to have a generic description of the problem and area: scope, concepts and technology and/or a literature review (state-of-the-art). In case of a practical project, there should also be described the tools and the justification for their use.

Usually, this chapter is divided in multiple sections, to complement the topics.

trabalhos semelhantes já realizados, métodos de análise de dados, até discussões sobre visualização de dados e importância da gestão de dados no mercado atual

2.1 Nome do artigo

Texto...

Capítulo 3

Methodology

A seção de metodologia explora uma abordagem para o desenvolvimento do software, iniciando com a definição dos requisitos, passando pela definição do método de recebimento dos dados, e a seleção das tecnologias. O processo de desenvolvimento incorpora a criação de histórias de usuário, organizadas em um quadro Kanban para gerenciamento de tarefas, e à configuração dos repositórios. A documentação e reuniões regulares que acompanham o progresso, são uma constante em todas as fases para garantir que o projeto está ocorrendo como o esperado.

3.1 Definição dos requisitos

A definição precisa dos requisitos é fundamental para garantir que o sistema desenvolvido atenda às necessidades e objetivos. Os requisitos desse projeto foram classificados nas categorias funcionais e não funcionais, para garantir uma compreensão completa do que é esperado do sistema.

3.1.1 Requisitos Funcionais

Os requisitos funcionais desempenham um papel fundamental no desenvolvimento de sistemas, definindo as funções que um sistema ou componente de software deve ser capaz

de executar. Em essência, eles fornecem uma descrição das interações que o sistema terá com seus usuários ou com outros sistemas, especificando os serviços que o sistema deve fornecer.

Para garantir eficácia, os requisitos funcionais devem ser claramente definidos, sem ambiguidades, e serem mensuráveis, rastreáveis, completos e consistentes. Além disso, eles devem ser definidos levando em consideração as necessidades e objetivos do projeto, garantindo que o sistema desenvolvido seja não apenas tecnicamente sólido, mas também útil e relevante para seus usuários finais.

Dentro do contexto do sistema desenvolvido, está listado abaixo os requisitos funcionais do sistema e sua descrição, de forma a deixar claro, de forma funcional, o que o sistema deve fazer.

FR1 - O sistema deve permitir que um usuário acesse o sistema de forma segura com um email e senha

Dado que o sistema é para a visualização dos dados de operação de uma industria de estampagem, as informações disponibilizadas devem ser acessadas apenas por usuários previamente autorizados.

FR2 - O sistema deve permitir que um usuário veja as suas informações pessoais que são armazenadas nos sistema

Cada usuário que tiver acesso ao sistema terá registrado nele alguns de seus dados pessoais, como e-mail, cargo e tipo de acesso. Portanto, cada usuário deve ter acesso a suas informações pessoais que estão salvas no sistema.

FR3 - O sistema deve exibir em tempo real os valores lidos pelos sensores em cada uma das máquinas

Ao receber os dados enviados pelos sensores, os sistema deve exibir em tela os valores lidos, separado por tipo de sensor e máquina.

FR4 - O sistema deve armazenar um valor máximo ideal para cada para tipo de sensor utilizado

Cada sensor deverá ter um valor máximo ideal para o funcionamento. Ele servirá de parâmetro para entender se o valor lido pelo sensor indica um bom ou mal funcionamento da máquina.

FR5 - O sistema deve identificar sempre que um valor lido pelo sensor não estiver abaixo do valor ideal

Esse requisito refere-se à capacidade do sistema de detectar, de forma automática, toda vez que o sensor indicar um valor que não esteja abaixo dO limite pré definido. Isto é, se o valor ideal for X, e o sensor ler um valor maior ou igual a X, o sistema reconhecerá esta situação.

FR6 - O sistema deve registrar sempre que um valor lido pelo sensor não estiver de acordo com o valor ideal

Esse requisito implica que o sistema deve manter um registro de todos os momentos em que o valor detectado pelo sensor não estiver abaixo do valor ideal armazenado.

FR7 - O sistema deve mostrar em tela quando um valor lido pelo sensor não estiver abaixo do ideal

Sempre que o sensor detectar um valor abaixo do padrão ideal, o sistema deverá exibir um alerta na interface de forma que fique sempre visível para o usuário.

FR8 - O sistema deve mostrar no formato de notificação os registros de não funcionamento abaixo do valor ideal

Este requisito estabelece que o sistema deve apresentar aos usuários na forma de notificações quando o sensor ler um valor acima do ideal, para possibilitar que os usuários sejam informados, mesmo que posteriormente, sempre que um alerta for identificado.

FR9 - O sistema deve permitir que o usuário marque uma notificação como lida, de forma que ela não apareça novamente

Após ser notificado, os usuários deverão ter a capacidade de marcar essa notificação como "lida", garantindo que a mesma informação não continue a ser exibida repetidamente.

FR10 - O sistema deve exibir gráficos mostrando os valores lidos pelos sensores nos dias anteriores de forma agregada, separando por máquinas

Esse requisito assegura que os usuários possam visualizar, por meio de representações gráficas, as leituras dos sensores de dias anteriores de forma agregada. Estes gráficos devem ser categorizados por máquina, proporcionando uma análise detalhada do desempenho de cada equipamento ao longo do tempo.

FR11 - O sistema deve exibir nos gráficos uma análise estatística do funcionamento das máquinas, junto com o valor máximo de funcionamento ideal

Os gráficos devem oferecer uma análise estatística, mostrando os indicadores estatísticos dos dados agregados média, mediana, percentil 75 e média removendo os outliers. Juntamente com isso, o gráfico também mostrará o valor ideal, servindo como uma referência para avaliar o desempenho.

FR12 - O sistema deve permitir filtrar as informações exibidas em tela por máquinas

Os usuários devem ter a flexibilidade de selecionar e visualizar informações específicas para determinadas máquinas, permitindo que eles se concentrem em equipamentos específicos conforme a necessidade.

FR13 - O sistema deve permitir filtrar os gráficos exibidas em tela por data

O sistema deve oferecer a capacidade de os usuários filtrarem as exibições gráficas por datas específicas, permitindo análises temporais detalhadas e comparações entre diferentes

períodos.

FR13 - O sistema deve exibir os gráficos de parada das máquinas de forma a exemplificar a exibição desses dados

O sistema deve mostrar as paradas de máquina de acordo com os dados passados pelas planilhas com os dados. Dessa forma poderá ser exemplificado como ficaria as informações de parada das maquinas caso o sistema recebesse essas informações.

3.1.2 Requisitos Não Funcionais

Requisitos não funcionais são especificações que determinam as características de desempenho, usabilidade, confiabilidade e outras propriedades que o sistema deve possuir, ao invés de comportamentos específicos que ele deve demonstrar. Enquanto os requisitos funcionais descrevem o que um sistema deve fazer, os requisitos não funcionais especificam como o sistema deve realizar essas funções.

Estes requisitos são cruciais para garantir a satisfação do usuário e a eficácia operacional do sistema, desempenhando um papel fundamental na qualidade e na operação geral de um produto de software.

Os requisitos não funcionais podem ser de vários tipos, como usabilidade, desempenho, segurança, disponibilidade, manutenção e confiabilidade. Dentro do contexto do sistema desenvolvido, está listado abaixo os requisitos não funcionais do sistema e sua descrição, de forma a deixar claro o que foi levado em consideração no momento de desenvolver cada uma das funcionalidades do sistema.

NFR1 - Disponibilidade

O sistema deve possuir mecanismos de reconexão automática que se ativam quando problemas de conexão ou recebimento de dados dos sensores forem detectados, garantindo assim a continuidade no recebimento dos dados.

NFR2 - Segurança no acesso

O sistema deve implementar controles de acesso para que somente colaboradores autorizados tenham permissão para acessar os dados e funcionalidades pertinentes ao seu papel.

NFR3 - Segurança na rede

Para garantir a segurança da transmissão de dados, a conexão ao sistema deve ser estabelecida utilizando o protocolo HTTPS, que incorpora a camada de segurança TLS, protegendo assim os dados contra intercepções e alterações.

NFR4 - Transmissão em tempo real

O sistema deve processar e transmitir os dados enviados pelos sensores em uma arquitetura baseada em streaming. O atraso entre o envio do dado pelo sensor e sua visualização pelo usuário final deve ser inferior a três segundos.

NFR5 - Modularidade

A arquitetura do sistema deve ser modular, possibilitando a integração e a adição de novos componentes ou funcionalidades de maneira eficiente e sem comprometer o funcionamento das partes já existentes.

NFR6 - Manutenibilidade

Priorizando a longevidade e facilidade de manutenção, o sistema deve ser desenvolvido seguindo boas práticas de programação e os princípios do SOLID. Isso facilitará futuras modificações, expansões e a correção de eventuais problemas.

NFR7 - Escalabilidade de sensores e máquinas

O design do sistema deve ser capaz de lidar com um crescente volume de sensores e máquinas, garantindo que não haja degradação de performance ou falhas quando a demanda por recursos aumentar.

NFR8 - Portabilidade

O sistema deve garantir compatibilidade com os principais navegadores web disponíveis no mercado. Além disso, a interface de usuário deve se adaptar bem em telas maiores como televisões, permitindo que o dashboard seja visualizado de forma clara em diferentes ambientes da fábrica.

NFR9 - Usabilidade

A interface do sistema e seus componentes devem ser projetados considerando princípios fundamentais de design de interação, garantindo que os usuários possam compreender e interagir com o sistema de maneira intuitiva e eficiente.

3.2 Método de coleta e armazenamento de dados

Dentro do contexto do projeto, a forma como os dados dos sensores são coletados e armazenados influenciam muito no funcionamento do sistema, pois é a partir deles que todo o sistema é estruturado. Dessa forma foi utilizado como base um protocolo desenvolvido em outro projeto, que transmite todas as informações necessárias para o contexto desse projeto. Dentro do sistema em questão, ficou a responsabilidade de implementar o decodificador para o determinado protocolo.

O protocolo utilizado foi desenvolvido dentro do mesmo contexto desse projeto, dentro do IPB, para atender uma demanda da mesma empresa, portanto ele se tornou a opção mais ideal, otimizando a comunicação entre os sensores e o sistema. O formato é estruturado de forma a representar as informações pertinentes à máquina, ao tipo de comunicação, ao sensor e ao significado dos dados transmitidos, seguindo o seguinte formato:

Machine ID (2 bytes)

O campo *Machine ID* é responsável por identificar a máquina em questão e está dividido em dois subcampos:

- **High (bytes de ordem superior)**: Representa o tipo da máquina. Exemplos de valores possíveis são: prensa, torno, robot, tapete, entre outros.
- **Low (bytes de ordem inferior)**: Identifica o número da máquina.

Type (1 byte)

O campo *Type* indica o tipo de mensagem e pode assumir os seguintes valores:

1. Publish
2. Request to publish

Sensor ID (2 bytes)

O campo *Sensor ID* fornece detalhes sobre o sensor que está transmitindo os dados:

- **High (bytes de ordem superior)**: Representa a quantidade física sendo medida, como temperatura, velocidade, pressão, força, entre outros.
- **Low (bytes de ordem inferior)**: Indica o número do sensor.

Meaning of Data (2 bytes)

O campo *Meaning of Data* fornece informações sobre o tipo e significado dos dados:

- **High (bytes de ordem superior)**: Tipo dos dados:
    1. Not defined
    2. Normal
    3. Raw data

4. Alarm

- **Low (bytes de ordem inferior)**: Significado dos dados, que varia de acordo com o equipamento. Exemplos incluem:
  - Oil critical temperature
  - Check oil temperature
  - Oil pressure

Length (2 bytes)

O campo *Length* indica o número de bytes subsequentes no pacote.

Data

Este campo representa os dados transmitidos pelo sensor. A especificação exata do que os dados representam deve ser definida e normalizada, conforme indicado pela notação (*).

3.3 Processo de desenvolvimento do software

Com a definição clara dos requisitos do sistema e da forma como os dados lidos pelos sensores são transmitidos, foi definido o processo de como o projeto seria desenvolvido. Esse processo de desenvolvimento passa pela definição das histórias de usuário, organização das atividades, organização da documentação, configuração dos repositórios no GitHub e reuniões periódicas com o professor e com a empresa para discutir o andamento.

3.3.1 Histórias de Usuário

No desenvolvimento ágil de software, uma das abordagens mais centradas no usuário entendimento das funcionalidades do sistema e dos requisitos é a utilização de *histórias de usuário* (ou *user stories* em inglês). Estas são descrições curtas, simples e informais

do ponto de vista de um usuário final, capturando o que eles necessitam ou desejam fazer no software.

A estrutura típica de uma história de usuário é: "Como [tipo de usuário], eu quero [uma ação] para que [um benefício/resultado]". Esta estrutura ajuda a manter o foco nas necessidades e desejos do usuário, em vez de mergulhar prematuramente em soluções técnicas ou detalhes de implementação.

Além de serem uma ferramenta de comunicação entre os desenvolvedores e os stakeholders, as histórias de usuário facilitam a priorização das tarefas, ajudam na criação de critérios de aceitação e fornecem uma base para discussões interativas durante as reuniões de revisão e planejamento.

Em suma, as histórias de usuário servem como um meio eficaz de traduzir requisitos complexos em tarefas gerenciáveis e centradas no usuário, garantindo que o produto final atenda às expectativas e necessidades dos seus utilizadores.

Com essa definição de historias de usuário, foi definido os seguintes itens que traduzem os requisitos em tarefas para o desenvolvimento do projeto:

1. As a user, I must be able to log in with my credentials to use the system.

2. As a user, I should be able to view my personal information on a profile page to manage the data the system holds about me.

3. As a user, I should be able to view in real-time values of the machines to detect relevant variations in operation more quickly.

4. As a user, I should be able to view the historical values of the sensors aggregated in charts of the machines to monitor the status over time.

5. As a user, I should be able to filter the dashboard information by machine and by date to view data according to my needs.

6. As a user, I should be alerted when a sensor reads a value that exceeds the ideal parameter so I can take necessary actions as quickly as possible.

7. As a user, I should be able to view system notifications to be alerted about machine operation alerts.

8. As a user, I should be able to view the machine downtime records for a better and more organized view of the recorded machine downtimes.

9. As a user, I should be able to view the system information (dashboards) on different screen sizes so that I can display the information in different contexts.

Cada uma dessas historias de usuário foi detalhada melhor na organização das tarefas, incluindo uma descrição mais completa, possíveis regras de negócio, quais requisitos, funcionais e não funcionais, ela se referencia, e também critérios de aceite. A organização das atividades está detalhada na seção 3.3.2.

3.3.2 Organização das tarefas

O método Kanban, originário do sistema Toyota de produção, tornou-se uma ferramenta popular e eficaz para gerenciamento e organização de fluxos de trabalho. A palavra "Kanban"é de origem japonesa e pode ser traduzida como "cartão visual"ou "sinalização". No contexto de gestão de projetos, Kanban refere-se a um sistema visual de gestão que destaca o fluxo de trabalho e as tarefas em diferentes estágios de processo. A essência do Kanban é visualizar todo o fluxo de trabalho, desde as tarefas que ainda não foram iniciadas até aquelas que foram concluídas. Esta visualização permite identificar gargalos e ineficiências, otimizando assim o processo. Para a organização das atividades desse projeto, o método Kanban foi adotado como uma estratégia para garantir uma progressão eficiente e sistemática do trabalho.

Para a implementação do método Kanban, foi escolhido o Notion como ferramenta. A escolha deste software deve-se à sua flexibilidade e capacidade de personalização, permitindo a criação de um board Kanban que se adapta especificamente às necessidades do projeto. Além disso, o Notion oferece uma interface intuitiva para a construção documentação, que está melhor detalhada na seção 3.3.3.

O board Kanban foi estruturado em cinco colunas, cada uma representando um estágio distinto no fluxo de trabalho:

- **Backlog:** Esta coluna contém todas as tarefas e atividades identificadas, mas que ainda não foram iniciadas. É um repositório de tudo o que precisa ser feito, mas que ainda não tem uma data definida para começar.
- **To Do:** As tarefas que estão nesta coluna estão prontas para serem iniciadas. Elas foram retiradas do Backlog, detalhadas e têm prioridade para serem iniciadas em breve.
- **Stopped:** Aqui estão as tarefas que foram iniciadas, mas por algum motivo tiveram que ser interrompidas, desde mudanças de prioridade, necessidade de alguma valição ou limitação técnica.
- **In Progress:** Esta coluna contém as tarefas que estão atualmente em execução. A movimentação para esta coluna indica que o trabalho está ativamente sendo feito na tarefa.
- **Done:** Assim que é finalizado o desenvolvimento de uma tarefa ela é dada como concluída, e portanto, movida para esta coluna. Representa o sucesso na finalização da atividade, e serve como um registro de todos os itens concluídos.

A estruturação destas colunas oferece uma visão clara do status de cada atividade e ajuda a identificar rapidamente onde estão os gargalos, facilitando a tomada de decisões e a priorização das tarefas.

A fim de definir melhor a implementação de cada história de usuário, cada card no board Kanban foi detalhado com uma descrição que inclui regras de negócio relevantes, referências a requisitos funcionais e não funcionais, critérios de aceitação e sub-tarefas. Antes que uma história de usuário possa ser movida para a coluna "In Progress", é essencial que esses campos sejam avaliados para assegurar um entendimento do escopo da tarefa. Os critérios de aceitação desempenham um papel importante na verificação de que uma história atende a todas as exigências estabelecidas antes de ser marcada como concluída.

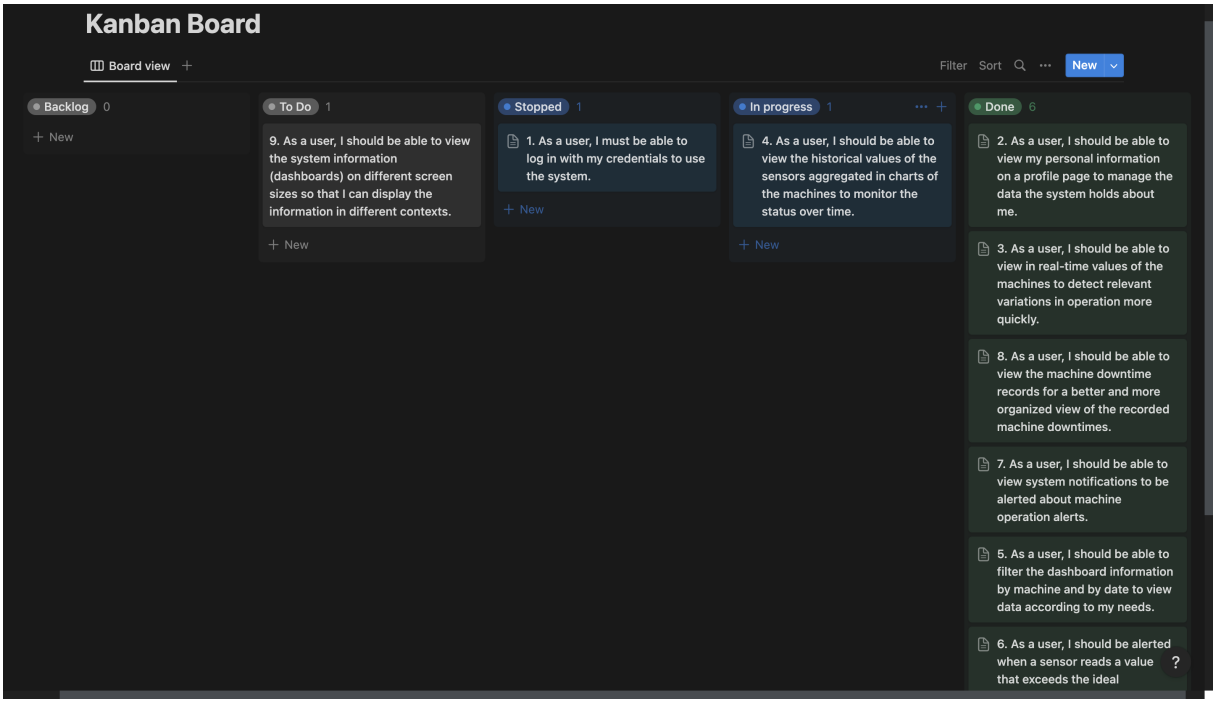

Figura 3.1: Board Kanban used to manager the tasks.

Para ilustrar este processo, é detalhado a história de usuário número 1.

1. As a user, I must be able to log in with my credentials to use the system.

- **Descrição:** Para garantir a segurança e a personalização da experiência do usuário, o sistema deve possuir uma funcionalidade de autenticação. O usuário deve inserir suas credenciais - geralmente um nome de usuário ou endereço de e-mail e uma senha - para acessar sua conta e as funcionalidades associadas a ela.

- **Regras de Negócio Relevantes:**

  – Os usuários não podem acessar o sistema sem autenticação.

  – As tentativas de entrar no sistema devem ficar armazenadas no banco de dados.

  – As senhas devem ser armazenadas de forma segura, utilizando técnicas como hashing.

- **Referências a Requisitos:**

  - **Funcionais:** FR1
  - **Não Funcionais:** NFR2, NFR3, NFR6, NFR9

- **Critérios de Aceitação:**
  - O sistema deve apresentar uma tela de login clara e intuitiva.
  - Após inserir as credenciais corretamente, o usuário deve ser redirecionado para a página inicial do sistema (/dashboard).
  - Se as credenciais estiverem incorretas, o usuário deve receber uma mensagem de erro clara.

- **Sub-tarefas:**
  - Desenhar a interface da tela de login.
  - Construção da interface de login no repositório.
  - Implementar a lógica de criação de3 usuário com hash de senha.
  - Implementar a lógica de autenticação no backend com JWT Token.
  - Testar a segurança e eficácia da funcionalidade de login.
  - Testar se a API está retornando os códigos HTTP corretamente.

3.3.3 Documentação do projeto

A documentação do projeto foi feita na ferramenta Notion. A escolha para este projeto foi fundamentada pois o Notion se destaca por sua flexibilidade, permitindo uma configuração adaptável dos texto para atender a vários tipos de necessidades. A interface intuitiva torna a inserção e atualização de informações um processo simples, que pode ser executado por ser uma plataforma acessível na internet por qualquer navegador.

Dessa forma, a documentação foi elaborada adotando uma abordagem estruturada para garantir que todas as características e funcionalidades fossem devidamente catalogadas. A base dessa documentação é uma tabela, onde cada entrada corresponde a uma

funcionalidade ou a uma específica parte do sistema, sendo denominada de acordo com o nome da característica em questão.

Cada item da tabela se expande em uma página independente, com detalhes descritivos. Esta organização permite o entendimento em cada aspecto do sistema, pois foi sendo construído à medida que o sistema era desenvolvido, refletindo assim as adições e mudanças mais recentes.

Para facilitar a navegação e busca por informações específicas, foi incorporado uma coluna de tags na tabela. Estas tags servem para categorizar as funcionalidades, e também um mecanismo eficiente de busca, permitindo que os usuários identifiquem rapidamente os aspectos relevantes do sistema.

Além disso, o design interno de cada página é fundamentado na estrutura do markdown, para assegurar que o conteúdo da documentação seja apresentado de forma clara, estruturada e esteticamente agradável, facilitando a compreensão e a absorção das informações por parte do leitor.

3.3.4 Configuração dos repositórios

A escolha da plataforma para gerenciamento dos repositórios do projeto foi o GitHub, uma das mais renomadas e amplamente adotadas plataformas de controle de versão baseada em Git disponíveis atualmente. A decisão de utilizar o GitHub foi pautada em alguns fatores. Primeiramente, a plataforma oferece uma interface intuitiva e um conjunto robusto de ferramentas que facilitam o acompanhamento do progresso do código, bem como a colaboração entre diferentes membros da equipe. Além disso, o GitHub é amplamente reconhecido por sua comunidade ativa, o que se traduz em uma vasta gama de recursos, tutoriais e suporte disponível, essencial para resolver possíveis dúvidas e desafios. Ademais, a integração com outras ferramentas e plataformas é facilmente realizável caso fosse necessário, permitindo um fluxo de trabalho contínuo e otimizado. Por fim, o compromisso do GitHub com a segurança, garantindo a integridade do código e dos dados do projeto, reforçou nossa decisão em adotá-lo como a solução de controle de versão para

o projeto desenvolvido.

Nesse contexto, os repositórios criados para esse projeto foram `backend`, que armazena o código referente a API do sistema, outro chamado `frontend`, que armazena o código do dashboard que é exibido na web, e outro chamado `iot_sensors_data_aggregation`, responsável por armazenar o código que realiza a agregação dos dados recebidos pelos sensores.

3.3.5 Reuniões periódicas

O desenvolvimento do projeto foi acompanhado por reuniões para garantir seu alinhamento com os objetivos do projeto. Reuniões semanais com o professor orientador foram estabelecidas, garantindo uma constante revisão, momento para tirar dúvidas, e realizar o detalhamento de atividades. Estes encontros proporcionaram um feedback contínuo, possibilitando a correção de trajetória e o foco no progresso desejado para o projeto. Paralelamente, reuniões mensais foram conduzidas com a empresa financiadora do projeto, para apresentar o que estava sendo construído, pegar feedbacks, orientações sobre funcionalidades desejadas, e também entender melhor como funciona a empresa e suas necessidades.

Esses encontros desempenharam um papel fundamental na integração entre a pesquisa acadêmica e as necessidades práticas da indústria, buscando que as soluções desenvolvidas se mantivessem pertinentes e aplicáveis ao contexto empresarial. Este sistema de supervisão foi crucial para manter o projeto em seu curso, balanceando as necessidades acadêmicas com a aplicabilidade industrial dentro do contexto da empresa.

3.4 Tecnologias

A seleção de tecnologias foi realizada para atender a requisitos específicos de escalabilidade e sustentabilidade a longo prazo. Dado que o sistema é primariamente voltado para armazenamento e gestão de dados, foi antecipada sua utilização como referência para futuros

projetos de características similares. Por isso, a ênfase recaiu sobre tecnologias modernas, amplamente reconhecidas e com robusto suporte na comunidade de desenvolvimento.

Nesse contexto, o MongoDB foi escolhido como nossa solução de banco de dados não relacional, devido à sua flexibilidade e performance. Python foi adotado como linguagem para o backend, devido à sua versatilidade e ampla biblioteca. O framework FastAPI, por sua vez, foi empregado para a elaboração da API, graças à sua eficiência e facilidade de integração. No âmbito do frontend, a linguagem JavaScript foi complementada pelo framework NextJs, reconhecido por sua otimização e recursos avançados. Para garantir uma integração fluida e modulada dos componentes do sistema, foi utilizado ao uso de containers Docker, enquanto o gerenciamento eficiente do servidor web foi assegurado pelo NGINX.

3.4.1 MongoDB

Ao selecionar uma plataforma de banco de dados, optou-se pelo MongoDB, um sistema de banco de dados não relacional projetado para adaptar-se com flexibilidade às mudanças no formato dos dados que são armazenados. O MongoDB proporciona facilidade na manipulação do data lake em variados contextos. Sua documentação meticulosamente estruturada, aliada à vasta gama de conteúdo disponível online, provou ser inestimável para a aquisição de conhecimento.

Distintivamente, o MongoDB apresenta vantagens como a capacidade de suportar consultas distribuídas e paralelas, otimizando o processamento de requisições intrincadas em cenários com densidade significativa de dados. Adicionalmente, sua compatibilidade com uma ampla variedade de ferramentas de análise de dados estabelece um precedente promissor para evoluções futuras do projeto.

3.4.2 Python

No desenvolvimento do backend, optou-se pela utilização da linguagem Python, amplamente reconhecida por sua versatilidade, legibilidade e adaptabilidade em diversos contextos de aplicação. Python, sendo uma das linguagens mais populares e amplamente aceitas no mundo acadêmico e industrial, apresenta uma vasta biblioteca padrão e suporte comunitário robusto. A rica gama de materiais educativos, que abrange desde tutoriais detalhados até extensos fóruns de discussão, foi essencial para o aprendizado.

A sintaxe intuitiva de Python favorece a rápida prototipagem e desenvolvimento, ao passo que a ampla gama de frameworks e bibliotecas disponíveis potencializa sua aplicação em diversos aspectos, desde análise de dados até desenvolvimento web. Essas características intrínsecas, somadas à flexibilidade e eficiência da linguagem, consolidam a decisão de adotar Python como a linguagem central para o backend neste projeto de mestrado.

3.4.3 FastAPI

Na etapa de implementação do backend, decidiu-se empregar a linguagem Python, aliada ao framework FastAPI. Esta escolha foi motivada, em grande parte, pela eficácia e elevado desempenho oferecido pelo FastAPI. Este framework se destaca por incorporar a biblioteca ASGI (Asynchronous Server Gateway Interface), uma interface que otimiza a gestão de solicitações, ao aproveitar ao máximo a execução assíncrona, garantindo respostas mais ágeis e precisas. Uma característica importante do FastAPI é sua documentação abrangente e bem elaborada, que ser como uma ferramenta fundamental ao processo de aprendizado e desenvolvimento.

3.4.4 NextJs

Para a arquitetura do frontend, optou-se pelo uso do Next.js, um framework que aprimora significativamente a interação com o React, tal como enfatizado na própria documentação oficial do React. O suporte comunitário é uma de suas principais características, sendo

amplamente complementado por uma oferta de materiais educativos disponíveis na internet - desde tutoriais a artigos de blog e vídeos instrucionais - os quais contribuíram de maneira essencial para o processo de aprendizado.

No escopo desse projeto, paralelamente ao Next.js, incorporou-se o TypeScript, que por sua natureza de tipagem estática, proporciona uma manutenção do código mais intuitiva, incrementando sua legibilidade, simplificando sua compreensão e gerenciamento.

A coerência entre Next.js e TypeScript estabelece um ambiente de desenvolvimento altamente eficaz. Enquanto o Next.js promove uma experiência de desenvolvimento mais fluída e de alto desempenho, o TypeScript fortalece a segurança e a produtividade, graças à sua tipagem rigorosa. Estes fatores justificam a escolha da combinação de Next.js e TypeScript para a concretização deste projeto.

Além do Next.js e TypeScript, o Material UI 5 foi integrado ao projeto como biblioteca de design de interface do usuário. Este conjunto de componentes React, baseado no padrão de design Material Design da Google, oferece uma ampla gama de elementos de interface já estilizados e de fácil implementação. Além da economia de tempo no desenvolvimento de componentes desde o início, a biblioteca proporciona uma experiência de usuário coesa e moderna. O uso do Material UI 5 também contribui para a padronização do design em todo o projeto, assegurando uma experiência de usuário mais intuitiva e agradável. Portanto, a adição deste recurso complementa eficazmente as capacidades já robustas oferecidas pela combinação de Next.js e TypeScript, tornando o ambiente de desenvolvimento ainda mais rico e produtivo.

3.4.5 Docker

Para a orquestração e gerenciamento do ambiente de desenvolvimento e produção, adotou-se o Docker como ferramenta de containerização. O Docker, amplamente reconhecido no universo de desenvolvimento de software, possibilita encapsular aplicações e suas dependências em containers, garantindo uniformidade, reprodutibilidade e isolamento entre os ambientes. Esta abordagem simplifica significativamente os processos de integração, teste

e implantação, uma vez que os containers podem ser movidos de forma transparente entre diferentes ambientes e plataformas.

A ampla documentação disponível, juntamente com uma comunidade ativa, proporcionou um entendimento claro e facilitou a adoção desta tecnologia. Além disso, a flexibilidade e eficiência proporcionadas pelo Docker, ao minimizar os conflitos de dependências e garantir que a aplicação funcione consistentemente em diversos contextos, foram fatores decisivos para sua escolha neste projeto.

3.4.6 NGINX

Para a parte de gestão das solicitações web, adotou-se o NGINX como servidor web. O NGINX é reconhecido por sua alta performance, confiabilidade e flexibilidade, sendo uma escolha adequada em ambientes de produção que demandam baixa latência, tratamento eficiente de um grande número de conexões simultâneas, e com capacidade de servir conteúdo estático de maneira extremamente rápida. Essas características tornam-no particularmente adequado para sistemas que visam escalabilidade e robustez.

A vasta documentação e os extensos recursos da comunidade foram essenciais para aprofundar o entendimento e aplicar as melhores práticas no contexto do projeto. Ao se considerar a necessidade de uma entrega consistente e otimizada do conteúdo ao usuário final, bem como uma configuração segura e eficaz do proxy, o NGINX emergiu como a escolha preeminente para esta dissertação de mestrado.

Capítulo 4

Arquitetura do sistema

Neste capítulo, é abordado em detalhe a arquitetura e a estrutura adotada para a construção dos componentes do sistema. É importante compreender que o sistema foi concebido como um conjunto de módulos, com cada um desempenhando funções específicas, e quando operados em conjunto, esses módulos resultam na realização dos propósitos pensados para o sistema.

O sistema é estruturado fundamentalmente em camadas distintas, o backend, o frontend, e o banco de dados.

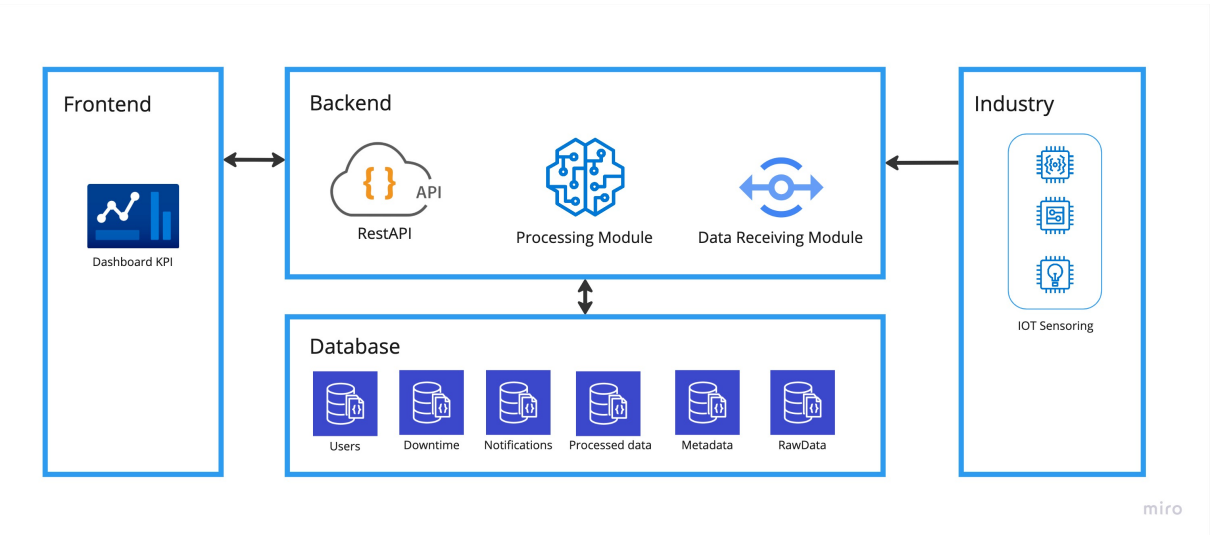

Figura 4.1: System architecture.

O backend funciona como o núcleo do sistema. Sua principal função é a de receber os dados, processá-los conforme as regras estabelecidas nos requisitos e histórias de usuários e, armazená-los de maneira segura no banco de dados. Além das funções de armazenamento e processamento, ao backend também é atribuída a responsabilidade de disponibilizar esses dados por meio de uma interface de programação de aplicações (API), que pode ser acessada utilizando métodos HTTP. Esta API age como um intermediário entre a lógica central do sistema e as interfaces com as quais o usuário final interage, o frontend.

Por outro lado, o frontend é caracterizado como a interface visual que o usuário final acessa. Serve como meio pelo qual os usuários interagem com o sistema, enviando e recebendo informações. Esta camada é projetada para acessar, recuperar e apresentar os dados processados e armazenados pelo backend de uma maneira intuitiva e amigável, utilizando princípios de design e data visualization.

Como pode ser visto na figura 4.1, na planta industrial, onde as máquinas com os sensores se encontram, os sensores enviam os dados para o sistema, que recebe eles por meio do modulo de recebimento de dados e os armazena no banco de dados. O modulo de processamento acessa os dados armazenados para realizar a agregação, e a API gerencia o acesso ao banco de dados, disponibilizando as informações para os usuários no frontend.

Nas seções seguintes, cada um desses componentes será explorado mais profundamente, passando por suas especificidades e interações.

4.1 Arquitetura backend

Nessa seção é abordado o funcionamento do backend. Ele está dividido em três partes, o module de recebimento dos dados dos sensores, o modulo de processamento onde é feita a agregação e analise estatística dos dados, e a API que gerencia o acesso as informações por meio de requisições HTTP.

Em relação a organização dos repositórios, a API e o modulo de recebimento de dados

ficam no mesmo repositório, facilitando a comunicação entre eles. Já o modulo de processamento está em um repositório a parte, sendo a sua única função sendo ler o banco de dados, processar os dados, e armazenar os resultados.

4.1.1 Modulo de Recebimento de dados

Para o modulo de recebimento de dados, inicialmente, destaca-se a classe `SensorConnection`, cuja principal função é gerenciar a conexão. Esta classe foi projetada para lidar com tarefas essenciais, como a inicialização e manutenção da conexão. Essa classe faz a transmissão dos dados recebidos à uma função designada `save_data_func`, assegurando que os dados sejam encaminhados para manipulação apropriada.

Na próxima parte da arquitetura, é utilizada a classe `IotSensorConnection`, que se origina da interface `IotSensorConnectionInterface`. Esta interface foi criada para garantir a adaptabilidade do sistema, facilitando a integração de diferentes tipos de recebimentos de dados, como por exemplo, uma classe destinada a gerar dados dos sensores em um ambiente de desenvolvimento, onde não há acesso ao sensor real. A classe `IotSensorConnection`, quando instanciada, é encarregada de estabelecer a conexão, e criar uma nova thread que opera como um ouvinte ativo, monitorando a chegada de novas informações. Ao perceber a recepção de novos dados, a classe direciona estas informações para uma terceira entidade, a qual detém a responsabilidade de aplicar as regras de negócio.

Esta terceira entidade é a classe `SensorsRepository`, que quando acionada com dados oriundos dos sensores, tem a responsabilidade de avaliar a informação com base nos parâmetros estabelecidos, decidindo se é imperativo acionar um alerta, e tornar os dados do sensor acessíveis via API, garantindo que esses dados estejam disponíveis para serem transmitidos em tempo real, via stream, para todos os usuários conectados. Além do mais, o dado é salvo no banco de dados, especificamente na coleção de dados brutos do data lake, `Raw Data`. Uma vez salvos no banco de dados, estes dados brutos estão disponíveis para serem processados pelo módulo de agregação.

A disponibilização dos dados pela classe `SensorsRepository` acontece por meio da instancia da classe `SensorValue`, que com o método `update_current_sensor_value` atualiza os dados em memoria que são acessados pelos usuários conectados.

O diagrama que mostra a organização dessas classes pode ser visto na figura 4.2. Este design assegura que os dados brutos dos sensores sejam efetivamente recebidos, avaliados e armazenados.

No capítulo 5, é aprofundado nos detalhes de implementação desse módulo.

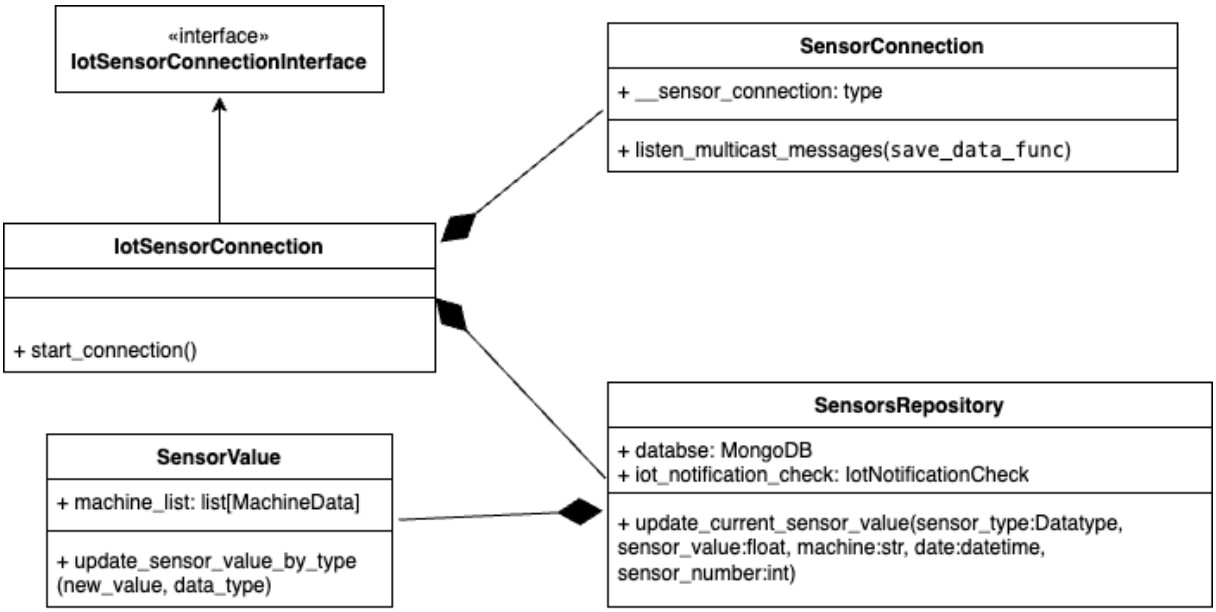

Figura 4.2: Module to receive sensor data.

4.1.2 Modulo de processamento

O módulo de processamento de dados foi desenvolvido para garantir que os dados brutos coletados sejam processados, fornecendo a análise estatística que é exibida para os usuários.

O código é executado quando é invocada uma função específica encarregada de realizar uma série de operações. Primeiramente, uma lista é constituída contendo as coleções do banco de dados responsáveis pelo armazenamento tanto dos dados brutos quanto dos

dados já processados. Simultaneamente, uma segunda lista é gerada, representando as máquinas que enviaram informações para o sistema.

Com essas listas em mãos, inicia-se um procedimento iterativo. Para cada máquina identificada, os dados disponíveis são lidos, submetidos a um processo de análise estatística, após o qual os resultados obtidos são registrados na coleção de dados processados. Essa análise estatística adota o método do Box Plot.

O Box Plot, também conhecido como diagrama de caixa, é uma ferramenta gráfica utilizada para representar a variação de dados observados de uma variável numérica por meio de quartis. Na figura 4.3, pode ser visto o retângulo formado pelo primeiro quartil (Q1), mediana e terceiro quartil (Q3), que fornecem uma noção sobre a centralidade e dispersão dos dados, enquanto as "antenas"estendem-se para mostrar a amplitude completa dos dados, ajudando assim na identificação de possíveis outliers.

Ao adotar o Box Plot, o sistema garante uma compreensão robusta da distribuição dos dados, identificando não apenas tendências centrais, mas também variações e potenciais anomalias.

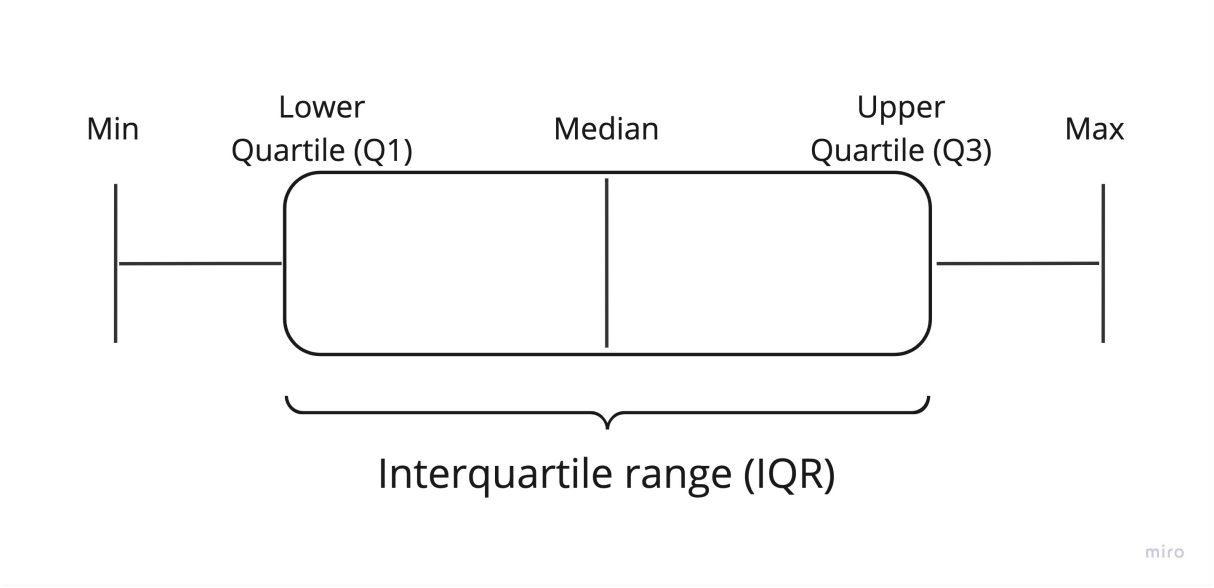

Figura 4.3: BoxPlot.

4.1.3 API

A API foi estruturada em pequenos sub-módulos, cada um focado em um contexto específico. Esta modularização assegura que cada parte da API tenha uma única responsabilidade. Em cada módulo, há uma segmentação composta por: a camada de *controller*, destinada a receber e gerenciar as requisições HTTP; a camada de serviço, que serve para processar a informação e aplicar as respectivas regras de negócio; e a camada de repositório, cujo papel é estabelecer uma ponte com o banco de dados, acessando e disponibilizando os dados necessários. A figura 4.4 representa a organização de pastas que foi utilizada para essa arquitetura.

Quando uma solicitação é enviada à API, a primeira interação acontece com a camada de *controller*. Uma vez recebida, essa requisição é direcionada à camada de serviço, onde as regras de negócio são aplicadas. A camada de serviço se comunica estreitamente com a camada de repositório, que detém a responsabilidade de acessar o banco de dados e trazer informações precisas, adequadas às demandas do módulo em questão.

Os módulos implementados na API com a lógica descrita são:

- **Downtime:** Responsável por gerenciar o acesso aos dados de paragem armazenados para teste no sistema.
- **IOT Sensors:** Responsável por gerenciar o acesso aos dados referentes aos sensores das maquinas na fabrica.
- **Notification:** Responsável por gerenciar o acesso as notificações geradas pelo sistema, e também as conexões web sockets para envio de notificações.
- **User:** Responsável por gerenciar o acesso aos dados dos usuários, assim como realizar as operações de login e logout.

Além dessas camadas modulares, existe uma área especifica na API para o armazenamento de códigos comuns a todos os módulos. Esta seção engloba diversas funções úteis, modelos de classes, valores constantes e configurações padrão. Tais elementos garantem uma maior coesão e reduzem a repetição de código, otimizando o desempenho

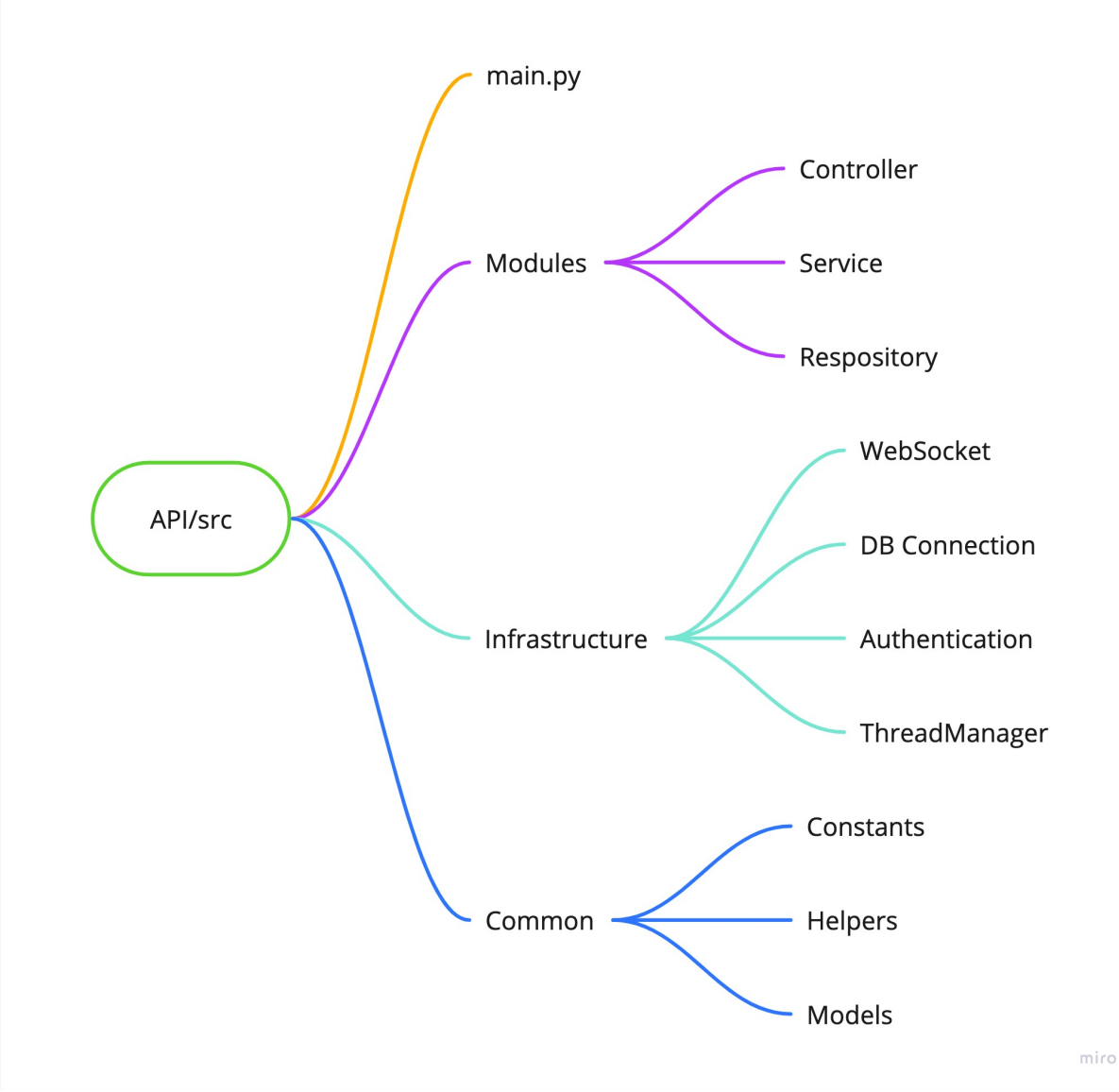

Figura 4.4: API Organization.

geral. Dentre as configurações padrão, merecem destaque o inicializador que estabelece o acesso ao banco de dados, middleware de autenticação, conexão web socket para envio de notificações, e o inicializador de novas *threads*. Este último é utilizada para operações assíncronas que são executadas em paralelo a operação da API, como aquelas executadas pelo módulo de recebimento de dados.

4.2 Arquitetura do frontend

Utilizando o *Next.js* como framework, o frontend segue uma estrutura básica já estabelecida pelo mesmo.

As rotas do sistema residem na pasta `pages`, alinhadas com as diretrizes do framework. Já os layouts que servem de base para cada página estão localizados na pasta `layouts`.

Os componentes *React* são a fundação de cada página e layout e estão organizados em uma camada específica, permitindo que sejam reutilizados em várias partes da aplicação.

Com a adoção do *Typescript*, modelos definem os tipos de estrutura de dados utilizados. Estes são mantidos na pasta `types`, estabelecendo contratos de formato de dados para o frontend. Isso minimiza erros e potencializa a eficiência no desenvolvimento.

A *Context API* do React é empregada para gerir dados nos componentes, permitindo o compartilhamento centralizado de informações, como pode ser visto na figura 4.5. Esta abordagem otimiza a maneira como os dados são acessados e distribuídos no sistema, otimizando a organização da arquitetura.

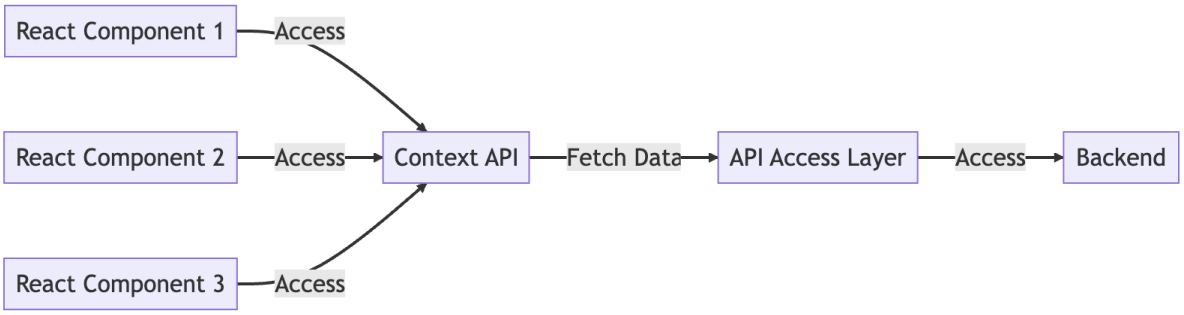

Figura 4.5: Frontend organization.

Existe uma camada específica para acesso externo, que administra a comunicação com a API e as conexões *WebSocket*. Esta é acessada apenas pelos contextos para atualização e recuperação de dados.

Por fim, há uma pasta dedicada para armazenar códigos recorrentes, contendo funções auxiliares, temas e *assets*, facilitando o desenvolvimento e manutenção ao proporcionar uma estrutura clara e coesa.

4.3 Containers

Containers são tecnologias que permitem isolar aplicações em ambientes específicos com todas as suas dependências, bibliotecas e configurações necessárias, sem a sobrecarga de máquinas virtuais completas. Isso garante que a aplicação funcione de maneira idêntica em diferentes ambientes, desde o desenvolvimento até a produção. Na figura 4.6 é possível visualizar como é o funcionamento dos containers dentro do sistema operacional do host.

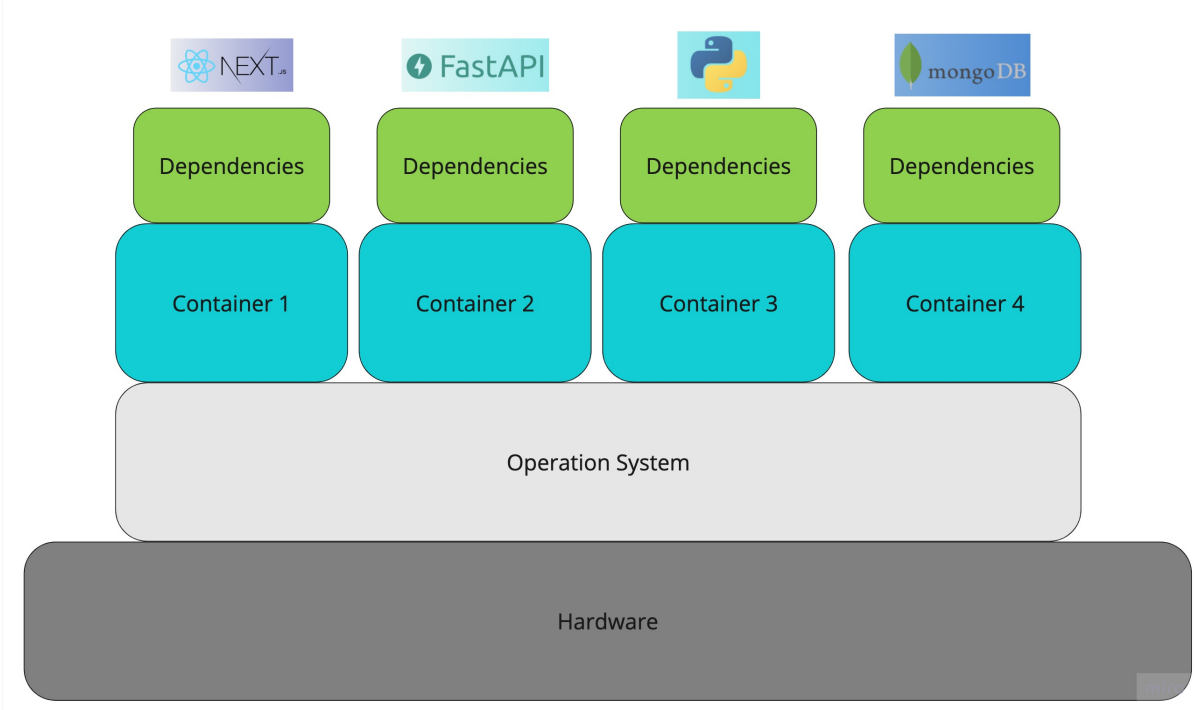

Figura 4.6: How container works.

Dentro do universo dos containers, o *Docker* foi a ferramenta selecionada para este projeto. Diversos fatores influenciaram essa decisão, incluindo uma documentação abrangente, uma comunidade ativa e a presença de uma ampla variedade de conteúdos disponíveis. Além disso, o Docker simplifica a definição, criação e execução de containers, tornando-se uma solução robusta para a implantação de aplicações.

O sistema adota diversos containers para organizar e gerenciar as várias partes da aplicação:

- **Frontend**: Um container dedicado ao frontend, construído com *NextJs*.
- **Backend**: Dividido em dois containers distintos, um abrange a API e o módulo de recebimento de dados, enquanto o segundo é voltado especificamente para o módulo de processamento.
- **Banco de Dados**: Um container para o banco de dados MongoDB, garantindo isolamento e eficiência na gestão dos dados.

A comunicação entre os containers é viabilizada através de uma rede *bridge* providenciada pelo Docker. Esta rede é uma interface de software criada no host que permite que containers comuniquem entre si e com o host, assegurando a conectividade necessária entre os diferentes módulos da aplicação. Com isso é adicionada uma camada a mais de segurança na aplicação, já que toda conexão externa deve ser feita por meio dessa rede. A conexão com rede externa é feita por meio de um web sever, explicado na seção 4.4.

Para garantir a persistência dos dados e evitar a perda de informações vitais, foi empregado o conceito de *volumes* do Docker na arquitetura do sistema. Volumes são espaços designados no sistema host que podem ser acessados e utilizados pelos containers. No contexto deste projeto, um volume foi especificamente configurado para o banco de dados MongoDB. Assim, mesmo que o container do banco de dados seja reiniciado ou removido, a base de dados se mantém intacta e disponível, devido à sua armazenagem no volume, que opera independentemente do ciclo de vida do container.

Com a necessidade de gerenciar múltiplos containers, configurações de rede e volumes de forma coesa e simplificada, foi adotado o *Docker Compose* na arquitetura do sistema. O

Docker Compose permite a definição e execução de aplicações multi-container usando um arquivo YAML. Esse arquivo contém todas as configurações necessárias para inicializar e interconectar os containers. Assim, ao invés de executar uma série de comandos para iniciar cada container individualmente, é possível, através do Docker Compose, iniciar todo o sistema com um único comando. Essa abordagem não apenas simplifica o processo de deploy e desenvolvimento, mas também garante que as configurações de rede e volume sejam consistentemente aplicadas em cada execução.

A utilização de containers no projeto trouxe vantagens. Primeiramente, garantiu a consistência entre os ambientes de desenvolvimento e produção. Adicionalmente, a modularização proporcionada pelos containers facilita a escalabilidade e manutenção do sistema, permitindo atualizações e alterações de forma ágil e segura a medida que o sistema for crescendo. Por último, a utilização de containers facilita a portabilidade do sistema, podendo ser executado em diversos tipos de servidores e sistemas, bastando ter a instalação do docker.

4.4 Web Server

Dentro da arquitetura proposta, com containers executando diferentes partes da aplicação, foi utilizado o *NGINX* para ser o intermediário no tráfego de requisições, assegurando a distribuição correta para cada container.

O método empregado para tal é o de proxy reverso. Em termos simples, o proxy reverso atua como uma interface entre o cliente e vários servidores, direcionando as solicitações dos clientes ao servidor adequado (no contexto desse projeto, os containers), e assim, otimizando o uso dos recursos e garantindo uma resposta mais rápida e eficiente.

No que se refere a requisições específicas, aquelas que envolvem retorno em formato de stream ou estabelecem uma conexão *WebSocket*, as configurações específicas foram feitas na configuração do *NGINX*, sendo essas detalhadas no capítulo 5, dedicado à implementação. Ao receber uma requisição, o servidor *NGINX* identifica, com base nela, qual container é o responsável pelo atendimento. Após essa identificação, são aplicadas as

configurações adequadas, e a requisição é direcionada ao container correspondente para obter a resposta. Esse workflow pode ser visto na figura 4.7.

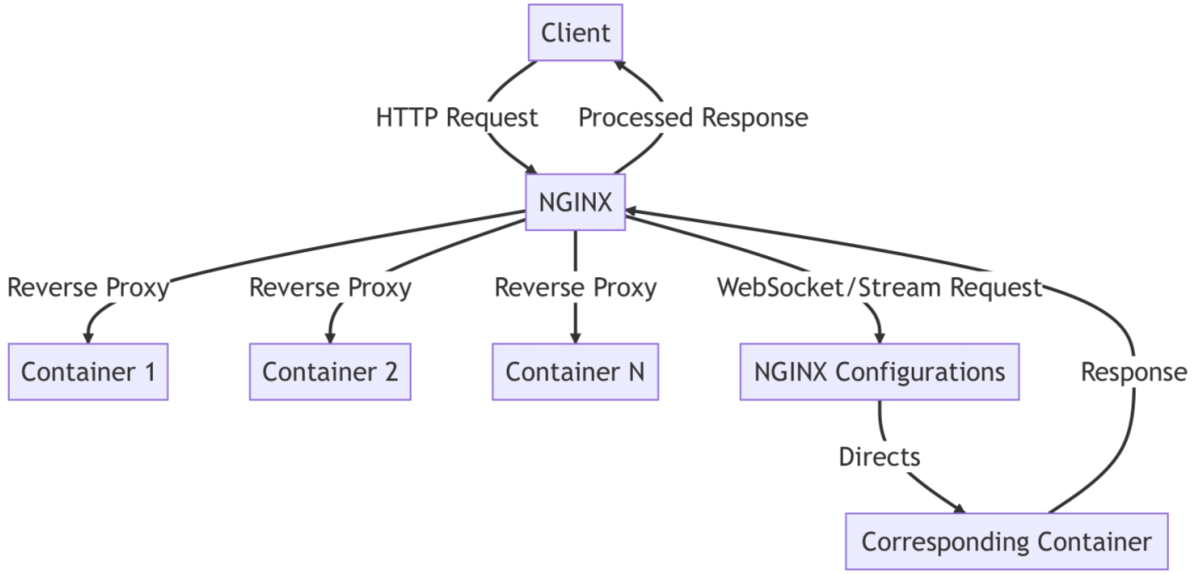

Figura 4.7: NGINX workflow.

A incorporação do *NGINX* trouxe alguns benefícios ao projeto. Um deles é a camada adicional de segurança: o *NGINX* limita o acesso direto aos containers, servindo como uma barreira contra tentativas de acesso não autorizado. Adicionalmente, com o NGINX, o processo de escalabilidade torna-se mais simples e eficiente, graças à capacidade inerente do servidor em atuar como um balanceador de carga. Este balanceador de carga distribui o tráfego de entrada entre vários servidores, assegurando que nenhum servidor fique sobrecarregado. Esta funcionalidade não só melhora a performance geral como também proporciona uma maior disponibilidade do sistema, já que, em caso de falha de um servidor, o tráfego pode ser direcionado a outro que esteja operacional.

Capítulo 5

Implementação

Após o capítulo onde a arquitetura do software foi detalhada, este capítulo é focado em explicar como tal arquitetura foi implementada, pois enquanto o primeiro descreve a estrutura e a organização, este foca nas ações técnicas adotadas para fazer essa estrutura funcionar.

Para uma análise mais estruturada e detalhada, este capítulo foi dividido em seções específicas para cada componente do sistema. São elas:

- **Implementação do banco de dados**: Esta seção abordará os detalhes técnicos do design do banco de dados, esquemas adotados e como as informações são armazenadas e recuperadas.
- **Implementação do módulo de recebimento de dados**: Esta seção detalhará como os dados são recebidos, validados e processados antes de serem armazenados e disponibilizados para os usuários.
- **Implementação da API**: Aqui, a estrutura da API será discutida, passando pelos endpoints fornecidos, a lógica por trás de cada um e as camadas utilizadas.
- **Implementação do módulo de processamento de dados**: É abordado o tratamento dos dados recebidos pelos sensores, e como é feito a analise estatística que gera as informações apresentadas nos gráficos.

- **Implementação do frontend**: Por fim, a interface com o usuário será discutida, explicando como os dados são estruturados apresentados e apresentados em tela.

5.1 Implementação do banco de dados

Dentro da implementação do sistema, o MongoDB foi usado para armazenar todas as informações do sistema. Este banco de dados, orientado a documentos, permitiu uma organização flexível dos dados, facilitando o armazenamento de diferentes dados que podem ser recebidos pelo modulo de recebimento de dados, e facilitando a criação de camadas de processamento. A estruturação dos bancos de dados e suas respectivas coleções foi pensada para facilitar tanto a inserção quanto a consulta de informações.

Em relação à organização dos dados, os seguintes bancos de dados foram criados:

- **Users**: Armazena informações referentes aos usuários. Possui coleções que registram tentativas de login, detalhes pessoais dos usuários e tokens associados a eles.

- **Notification**: Destinado às notificações do sistema. Atualmente, este banco contém apenas notificações associadas aos alertas das máquinas, gerados pelos dados recebidos dos sensores junto com os parâmetros armazenados.

- **Downtime**: Armazena duas coleções, uma com os dados lidos das planilhas de parada das máquinas, e outro com esses dados tratados. Esse banco de dados com essas coleções são apenas para simular como ficaria os dados de parada das maquinas, caso eles fosses inseridos no sistema.

- **Raw Data**: Este banco é dedicado ao armazenamento de dados brutos oriundos de diferentes sensores. Cada tipo de sensor, como os sensores de pressão, tem sua própria coleção, garantindo um agrupamento das informações que facilita a análise.

- **Processed Data**: Como o próprio nome sugere, armazena dados que já passaram por uma etapa de processamento. Assim, dados interpretados de diferentes sensores

são separados em coleções específicas, como os de pressão em uma e os de voltagem em outra.

- **Metadados**: Dedicado à armazenagem de metadados do sistema. Até o momento, a única coleção presente é a "AlertParameter", que reúne parâmetros utilizados para gerar alertas associados a cada sensor.

Com esta estruturação, busca-se não apenas organizar de forma lógica os dados, mas também otimizar operações de consulta e garantir uma expansão simplificada à medida que novas necessidades de armazenamento emergem no sistema.

A implementação do acesso ao banco de dados está detalhada na seção de implementação da API, em 5.4.2.

5.2 Implementação do modulo de recebimento de dados

No processo de implementação do sistema, uma das etapas essenciais foi o desenvolvimento de um módulo destinado ao recebimento de dados provenientes dos sensores IoT. Este recebimento é realizado por meio de uma conexão multicast, uma abordagem eficiente para lidar com a transmissão de mensagens a vários destinatários simultaneamente.

Esse modulo é responsável por estabelecer a conexão multicast para receber os dados, realizar a conversão dos dados recebidos de acordo com o protocolo pré definido, disponibilizar os dados para serem mostrados em tempo real para os usuários conectados, verificar se gera algum tipo de alerta (e se gerar, notificar os usuários sobre isso com a criação de uma notificação), e salvar as informações geradas no banco de dados.

5.2.1 Conexão e recebimento dos dados

A classe `SensorConnection` tem como principal responsabilidade criar um socket, manter-se conectada para receber mensagens e interpreta-las. A estrutura e o funcionamento desta

classe são detalhados a seguir.

A classe `SensorConnection` é iniciada com a criação de um socket IPv4 e UDP:

```
class SensorConnection:
    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Para garantir que o sistema esteja constantemente ouvindo mensagens multicast dos sensores, o método `listen_multicast_messages` foi definido dentro dessa classe. Ele invoca a criação da conexão e inicia o processo de leitura de mensagens, gerenciando ainda possíveis desconexões e reestabelecendo a ligação quando necessário:

```
    async def listen_multicast_messages(self, save_data_func):
        self.__create_connection()
        while True:
            await self.__start_read_messages(save_data_func)
            self.sock.close()
            time.sleep(1)
            self.__reconnect()
```

A função `__create_connection` tem o papel de estabelecer e configurar a conexão inicial com o grupo multicast, e dentro do loop infinito é inciado o recebimento das mensagens com o método `__start_read_messages`. Quando esse método é finalizado a conexão socket é fechada, e em seguida reconectada para depois voltar a fazer a leitura das mensagens. A chamada da função `time.sleep(1)` é utilizada para ter um pequeno intervalo entre uma chamada e outra caso e não realizar uma quantidade muito grande de chamadas caso esteja ocorrendo algum tipo de problema.

A seguir, cada uma das funções chamadas dentro desse método e detalhado.

Método create connection

```
    def __create_connection(self):
```

```
self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_address = ('', SENSOR_MULTICAST_PORT)
self.sock.bind(server_address)

multicast_group = SENSOR_MULTICAST
group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
self.sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
```

Inicialmente, o socket é configurado para permitir várias conexões em um único endereço. A opção `SO_REUSEADDR` é definida com o valor 1, permitindo que mais de um socket se ligue a um mesmo endereço, o que é especialmente útil em contextos de conexões multicast:

```
self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Após isso, o socket é vinculado a um endereço e porta multicast específicos. É importante ressaltar que o primeiro argumento na definição do endereço do servidor é deixado vazio. Esta abordagem garante que o sistema esteja conectando-se com todas as interfaces de rede disponíveis, proporcionando uma ampla cobertura de conexão:

```
server_address = ('', SENSOR_MULTICAST_PORT)
self.sock.bind(server_address)
```

Por fim, para se juntar efetivamente ao grupo multicast, algumas etapas são realizadas. O endereço IP multicast é primeiramente convertido para o formato binário com a chamada de `socket.inet_aton`. Em seguida, este endereço e o endereço local (representado por `socket.INADDR_ANY`) são empacotados em uma estrutura de dados por `struct.pack`. Esta estrutura é usado para especificar ao socket que ele deve se juntar a um grupo multicast em `self.sock.setsockopt`. A opção `IP_ADD_MEMBERSHIP` é definida

e a estrutura previamente criada é passada como argumento, concluindo a conexão com o grupo multicast:

```
multicast_group = SENSOR_MULTICAST
group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
self.sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
```

Essas operações garantem que o socket esteja configurado e conectado ao grupo multicast, pronto para receber mensagens de múltiplas fontes simultaneamente.

Método reconnect

```
def __reconnect(self):
    try:
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.__create_connection()
    except Exception as e:
        print(f"Error to reconnect: {e}")
```

Em situações em que a conexão com os sensores é interrompida, o método `__reconnect` é chamado para tentar estabelecer novamente a conexão, criando uma nova instância do socket e chamando novamente a função `__create_connection`, detalhada anteriormente.

Método start read messages

```
async def __start_read_messages(self, save_data_func):
    while True:
        try:
            data, address = self.sock.recvfrom(1024)
            result = self.__parse_multicast_message(data)
            if not type(result) == str:
```

```
            await save_data_func(result)
        except Exception as e:
            print(f"Error: {e}")
            break
```

Após as configurações realizads, as mensagens são continuamente lidas e processadas pela função `__start_read_messages`. Durante este processo, cada mensagem é processada pelo método `__parse_multicast_message`, e se estiver no formato correto, é passada para uma função que irá salvar e disponibilizar para API enviar por streaming para os usuários conectados.

Se ocorrer algum problema na execução desse método, ele é finalizado e volta para o `listen_multicast_messages`, onde o socket é fechado e uma nova conexão é estabelecida pelo método `__reconnect`.

Método parse multicast messages

```
def __parse_multicast_message(self, data):
        (machine_type_high, machine_number_low) =
            self.__parse_bytes(data[:2])

        message_type = data[2]

        if message_type == 2:
            return "Request to publish..."

        (physical_quantity_high, sensor_number_low) =
            self.__parse_bytes(data[3:5])

        (data_type_high, meaning_low) =
            self.__parse_bytes(data[5:7])
```

```
        message_dict = {
            'Machine': {
                'Type': str(machine_type_high)+". "+MACHINE_TYPE[machine_type_high],
                'Number': machine_number_low
            },
            'Type': str(message_type)+". "+ MESSAGE_TYPE[message_type],
            'Sensor': {
                'PhysicalQuantity': PHYSICAL_QUANTITY[physical_quantity_high],
                'Number': sensor_number_low
            },
            'MeaningOfData': {
                'DataType': str(data_type_high)+". "+DATA_TYPE[data_type_high],
                'Meaning': str(meaning_low)+". "+DATA_MEANING[meaning_low]
            }
        }

        return message_dict
```

Para interpretar e extrair informações da mensagem recebida do multicast, é crucial decodificar adequadamente a mensagem de acordo com o protocolo definido anteriormente. A implementação dessa decodificação é feita pelo método `__parse_multicast_message`. A função auxiliar `__parse_bytes` é utilizada para essa tarefa, dada uma sequência de bytes, a função interpreta os bytes utilizando a ordem big-endian (onde os bytes mais significativos vêm primeiro).

```
def __parse_bytes(self, bytes):
    data = int.from_bytes(bytes, byteorder='big')
    high_data = (data >> 8) & 0xFF
```

```
    low_data = data & 0xFF

    return (high_data,low_data)
```

Aqui, `data` contém o valor inteiro dos bytes fornecidos. O byte de ordem superior (High) é extraído deslocando o valor 8 bits para a direita e aplicando uma operação "END"(&), e o byte de ordem inferior (Low) é simplesmente obtido aplicando a operação "END"com `0xFF`.

Com a capacidade de interpretar os bytes, a função principal `__parse_multicast_message` pode começar a decodificação:

- Primeiro, ela extrai o tipo de máquina e o número da máquina dos dois primeiros bytes da mensagem.
- O terceiro byte da mensagem é então interpretado como o tipo da mensagem. Se o tipo da mensagem for 2, a função retornará diretamente uma solicitação para publicar.
- Os bytes 4 e 5 são interpretados como o ID do sensor, que contém a quantidade física sendo medida e o número do sensor.
- Os bytes 6 e 7 são usados para extrair o tipo de dados e seu significado.

A informação extraída é então organizada em um dicionário para representação clara e fácil acesso aos componentes individualmente:

```
message_dict = {
    'Machine': {
        ...
    },
    'Type': ...,
    'Sensor': {
```

```
        ...
    },
    'MeaningOfData': {
        ...
    }
}
```

Esta estrutura permite uma representação clara e modular da mensagem decodificada, tornando fácil a integração e utilização em outras partes do sistema. Sendo assim, o retorno do método `__parse_multicast_message` é utilizado como resultado da interpretação da mensagem multicast, e enviado para função recebida como parâmetro, `save_data_func`.

5.2.2 Verificação e disponibilização dos dados

No processo de recebimento dos dados, após abrir a conexão e os dados, é necessário verificar se estão no formato correto, se gera algum alerta, inserir no banco de dados e disponibilizar para os usuários conectados no sistema.

A classe `IotSensorConnection`, que implementa a interface `IotSensorConnectionInterface`, desempenha um papel principal neste módulo. Na sua inicialização, é estabelecida uma ligação com o repositório através da variável `self.__repository`. Além disso, é responsável pela conexão com o sensor é estabelecida por meio do `self.__sensor_connection`, explicada anteriormente na seção 5.2.1.

```
class IotSensorConnection(IotSensorConnectionInterface):
    def __init__(self, respository:SensorsRepository):
        self.__repository = respository
        self.__sensor_connection = SensorConnection()

    def start_connection(self):
        threadManager = ThreadManager()
```

```
        threadManager.start_async_thread(self.__start_connection)


    async def __start_connection(self):
        await self.__sensor_connection.
            listen_multicast_messages(self.__handle_iot_data)
```

Ao iniciar a conexão, utilizando o método `start_connection`, é criada uma nova thread por meio da classe `ThreadManager`. Esta thread invoca o método `listen_multicast_messages` da classe `SensorConnection` que foi detalhado na seção 5.2.1. É necessário criar uma nova thread pois como esse modulo está junto com a API, e é necessário que os doi processos funcionem ao mesmo tempo, uma nova thread foi necessária para o funcionamento em paralelo de ambos.

Verificação do formato dos dados

Para lidar com os dados recebidos, o método `__handle_iot_data` é passado como argumento para `listen_multicast_messages` (como o argumento save_func na classe que existe na classe `SensorConnection`).

```
async def __handle_iot_data(self, sensor_data:dict):
    sensor_model = self.__parse_sensor_data_to_sensor_model(sensor_data)
    await self.__repository.update_current_sensor_value(
        sensor_value = sensor_model.value,
        machine = sensor_model.machine,
        date = sensor_model.date,
        sensor_type = sensor_model.type,
        sensor_number = sensor_model.sensor_number
    )


def __parse_sensor_data_to_sensor_model(self, sensor_data:dict):
    value = 1
```

```
    machine = str(sensor_data["Machine"]['Number'])
        + sensor_data["Machine"]['Type']
    date = datetime.now()
    data_type = sensor_data["Sensor"]["PhysicalQuantity"]
    sensor_number = sensor_data["Sensor"]["Number"]
    return ConnectionModelToParse(
        date=date,
        machine=machine,
        sensor_number=sensor_number,
        type=data_type,
        value=value
    )
```

Este método tem a responsabilidade de receber os dados so sensor e converter para uma classe modelo, denominada `ConnectionModelToParse`, que utiliza o `Pydantic` para validar as informações. O uso do `Pydantic` é mostrado na seção **??**.

```
from datetime import datetime
class ConnectionModelToParse:
    def __init__(self,value:float,machine:str,
        date:datetime,type:Datatype,
        sensor_number:int):

        self.value = value
        self.machine = machine
        self.date = date
        self.type = type
        self.sensor_number = sensor_number
```

Após essa transformação, os dados são encaminhados para o repositório. O método `update_current_sensor_value` do repository é chamado para checar se o dado recebido

gera algum tipo de alerta, salvar no banco de dados, atualizar os dados em memoria, e realizar as verificações de notificação.

```
class SensorsRepository:
    def __init__(self):
        self.database = MongoDBIOT()
        self.iot_notification_check = IotNotificationCheck()
        self.__sensor_value = SensorValue()

    async def update_current_sensor_value(self, sensor_type:Datatype,
        sensor_value:float, machine:str, date:datetime,
        sensor_number:int):
        alert_type = await self.__get_alert_type(sensor_value, sensor_type)
        current_value = {"machine":machine,
            "value":sensor_value, "timestamp": date,
            "alert_type":alert_type.value,
            "sensor_number":sensor_number}
        result = await self.insert_value_into_database(current_value,
            sensor_type)
        new_id = result.inserted_id
        iot_data = IotData(
            alert_type=current_value["alert_type"],
            machine=current_value["machine"],
            timestamp=current_value["timestamp"],
            value=current_value["value"],
            id=PyObjectId(new_id),
            datatype=sensor_type,
            sensor_number=sensor_number
        )
```

```
        self.__sensor_value.update_sensor_value_by_type(
            iot_data,sensor_type)

        await self.iot_notification_check.check_iot_notification(
            iot_data)
```

Verificação de alertas

Dentro do método `update_current_sensor_value`, primeiramente é verificado o tipo de alerta gerado com a o método `__get_alert_type`. Esse método realiza a leitura do parâmetro de acordo do com tipo do sensor dentro dos metadados do sistema, em que o acesso é explicado em ..., e com ele verifica o status de alerta.

O stats de alerta, definido pela função `get_alert_status`, retorna como `OK` caso o valor do sensor seja menor que 90% do valor definido com parâmetro, retorna como `WARNING` caso esse valor esteja entre 90% e 100%, e retorna como `PROBLEM` caso o valor retornado pelo sensor seja maior que 100% do valor definido como parâmetro

```
def get_alert_status(self,sensor_value:int,
    alert_parameter:int)->AlertTypes:

    parameter = ((sensor_value/alert_parameter)*100)
    if parameter < 90:
        return AlertTypes.OK
    if parameter >= 90 and parameter < 100:
        return AlertTypes.WARNING
    if parameter >= 100:
        return AlertTypes.PROBLEM

async def __get_alert_type(self, sensor_value:float,
```

```
    sensor_type:Datatype)->AlertTypes:
    alert_parameter = await MetadataRepository()
        .get_sensor_alert_value(sensor_type)
    alert_type = self.get_alert_status(sensor_value,alert_parameter)
    return alert_type
```

Registro no banco de dados

Com a verificação dos alertas, todas as informações foram geradas, portanto já podem ser registradas no banco de dados. Para esse registro é usado o método `insert_value_into_database`.

```
async def insert_value_into_database(self, value:BaseIotData, type:Datatype):
    try:
        collection = sensor_name_to_raw_data_collection(type)
        return await self.database.insert_one(IOT_DATABASE,collection,value)
    except Exception as ex:
        print(ex)
        raise ex
```

Esse método utiliza da classe base do banco de dados com operações já definidas para realizar o registro. Dentro do método `update_current_sensor_value` do repositório, o retorno é utilizado para manter o ID registrado em memoria, importante para criar o objeto `IotData`, que é enviado para os usuários conectados, via stream, no passo seguinte.

```
current_value = {"machine":machine,
    "value":sensor_value,
    "timestamp": date,
    "alert_type":alert_type.value,
    "sensor_number":sensor_number}
result = await self.insert_value_into_database(current_value, sensor_type)
new_id = result.inserted_id
```

```
iot_data = IotData(
    alert_type=current_value["alert_type"],
    machine=current_value["machine"],
    timestamp=current_value["timestamp"],
    value=current_value["value"],
    id=PyObjectId(new_id),
    datatype=sensor_type,
    sensor_number=sensor_number
)
```

Uma informação importante a se destacar, é que o nome da coleção utilizada pelo método `insert_value_into_database` é definida de acordo com tipo de dado estabelecido, usando a função de ajuda `sensor_name_to_raw_data_collection`, explicada anteriormente em ....

Atualização dos dados em memoria

Com o tipo de alerta definido e os dados registrados no banco de dados, é utilizado a classe `SensorValue` para atualizar as informações na memória. Esse processo é feito por meio da chamada `__sensor_value.update_sensor_value_by_type (iot_data,sensor_type)` no método `update_current_sensor_value` do repository.

A classe `SensorValue` é responsável por gerenciar e atualizar os valores em memória. Nota-se que a mesma utiliza o padrão de projeto `Singleton`, assegurando a existência de apenas uma instância desta classe durante todo o ciclo de vida da aplicação, garantindo que so existe uma instancia armazenando as informações dos sensores.

```
class SensorValue(metaclass=Singleton):
    def __init__(self) -> None:
        self.machine_list:list[MachineData] = []


    def update_sensor_value_by_type(self, new_value: IotData, data_type: Datatype):
```

```
        is_new_machine = True

        for machine in self.machine_list:
            if machine.name == new_value.machine:
                is_new_machine = False
                is_new_sensor = True

                for index, sensor in enumerate(machine.sensor_data):
                    if sensor.datatype == data_type:
                        machine.sensor_data[index] = new_value
                        is_new_sensor = False
                        break

                if is_new_sensor:
                    machine.sensor_data.append(new_value)
                    break

        if is_new_machine:
            new_machine = MachineData(name=new_value.machine,sensor_data=[new_value])
            self.machine_list.append(new_machine)
```

No momento de sua inicialização, a classe `SensorValue` inicializa uma lista vazia, `machine_list`, que será responsável por armazenar os valores dos sensores organizados por máquina.

A atualização acontece pelo método `update_sensor_value_by_type`. Este método atualiza o valor do sensor na memória de acordo com seu tipo (`data_type`). O processo de atualização verifica primeiramente se a máquina associada ao sensor já existe na lista. Caso positivo, busca-se pelo sensor específico dentro dos dados da máquina e atualiza-se seu valor. Se o sensor não for encontrado, um novo é adicionado à lista de sensores da

máquina correspondente.

Por outro lado, se a máquina não for encontrada na lista `machine_list`, uma nova instância de `MachineData` é criada e adicionada à lista, contendo as informações da máquina e os dados do sensor recebido.

```
class MachineData(BaseModel):
    name:str = Field(...)
    sensor_data:list[IotData] = Field([])
```

Dessa forma, o repositório envia as informações para esse método, e com a verificação adequada, é mantido os dados mais atualizados em memoria, e disponível para ser utilizado pela API, possibilitando o acesso em tempo real dos dados dos sensores.

Verificação de notificação

Com o tipo de alerta verificado, a informação salva no banco de dados e o objeto `IotData` montado, a última tarefa do método `update_current_sensor_value` do repository é utilizar o singleton IotNotificationCheck para verificar as notificações em relação a operação das máquinas.

A classe `IotNotificationCheck` atua como um controlador de alertas para dados IoT. Ao receber dados IoT, ela verifica o estado do alerta e toma medidas apropriadas, seja adicionando ou removendo máquinas ou sensores da lista de alertas. Essa classe é essencial para monitorar e responder a eventos de alerta em tempo real, garantindo que os usuários associados sejam notificados de quaisquer anormalidades ou eventos importantes detectados pelos sensores IoT.

Por meio do método `check_iot_notification`, a classe verifica o tipo de alerta recebido pelo objeto IotData, se a máquina está em estado de alerta e se o sensor específico da máquina está em estado de alerta. Com base nessa verificação, o método toma uma das seguintes ações:

1. Coloca uma nova máquina em estado de alerta.

2. Coloca um novo sensor da máquina em estado de alerta.

3. Remove um sensor da máquina do estado de alerta. Se a máquina tiver apenas um único sensor em estado de alerta, a maquina é removida do estado de alerta

```
async def check_iot_notification(self, iot_data:IotData):
    is_alert_value = self.__is_alert_type_a_new_alert(
        iot_data.alert_type)
    machine_in_alert = self.__is_machine_in_alert_state(
        machine_name=iot_data.machine)
    machine_sensor_in_alert = self.__is_machine_sensor_in_alert_state(
        machine_in_alert,
        iot_data.datatype)

    is_machine_in_alert = machine_in_alert!=None

    if is_alert_value and is_machine_in_alert and (not machine_sensor_in_alert):
        await self.__put_new_machine_sensor_in_alert_state(
            machine_in_alert,
            iot_data.datatype)

    if is_alert_value and (not is_machine_in_alert):
        await self.__put_new_machine_in_alert_state(
            iot_data.machine,
            iot_data.datatype,
            iot_data.timestamp,
            iot_data.alert_type)

    if (not is_alert_value) and is_machine_in_alert and machine_sensor_in_alert:
        await self.__remove_machine_sensor_from_alert_state(
```

```
            machine_in_alert,
            iot_data.datatype,
            iot_data.timestamp)
```

O método `__put_new_machine_sensor_in_alert_state` é um método assíncrono privado que tem a responsabilidade de adicionar um novo sensor ao estado de alerta para uma máquina específica. Ele recebe dois parâmetros: `machine_in_alert`, que é uma instância da classe `MachinesSensorAlert` representando a máquina em questão, e `sensor_type`, que é uma instância do tipo `Datatype` representando o tipo de sensor que deve ser colocado em alerta.

```
class MachinesSensorAlert(BaseModel):
    id: PyObjectId = Field(default_factory=PyObjectId, alias="_id")
    machine:str = Field(...)
    sensors:list[str] = Field([])
    alert_type:str = Field(...)
    start_time:datetime = Field(...)
    sensors_historical:list[str] = Field([])
    is_in_alert:bool = Field(True)

    end_time:Optional[datetime|None] = Field(None)
    read_by:Optional[list[str]] = Field([])
```

Importante destacar que dentro dessa instância que é mantida em memória, o atributo `read_by` não é preenchido. Isso acontece pois esse atributo é usado para controlar os usuários que marcaram a notificação como lida, portanto é preenchida apenas no banco de dados. A implementação da parte de notificações que faz uso desse atributo pode ser lida na seção ....

A primeira etapa realizada por este método é identificar a posição (ou índice) da máquina dentro da lista de alertas `machines_alert` usando o método `index`. Uma vez

obtido o índice, o tipo do sensor é adicionado à lista de sensores em estado de alerta da máquina, representada pelo atributo `sensors`. Além disso, este sensor também é adicionado ao histórico de sensores em estado de alerta da máquina, indicado pelo atributo `sensors_historical`. Finalmente, a máquina atualizada (com o novo sensor adicionado às suas listas de alerta e histórico) é reinserida na lista principal `machines_alert` na mesma posição identificada anteriormente.

Este método, garante que sempre que um novo sensor entra em estado de alerta para uma máquina que ja tinha um sensor em alerta, as informações relevantes são adequadamente atualizadas e mantidas em memória, permitindo um acompanhamento em tempo real das condições de alerta de todas as máquinas monitoradas.

```
async def __put_new_machine_sensor_in_alert_state(
        self,
        machine_in_alert: MachinesSensorAlert,
        sensor_type:Datatype):
    index = self.machines_alert.index(machine_in_alert)
    machine_in_alert.sensors.append(sensor_type.value)
    machine_in_alert.sensors_historical.append(sensor_type.value)
    self.machines_alert[index] = machine_in_alert
```

Já o método `__put_new_machine_in_alert_state` é um método assíncrono privado cuja principal função é criar e registrar um novo estado de alerta para uma máquina específica. Este método é invocado quando uma máquina entra em estado de alerta pela primeira vez, o que significa que ainda não está presente na lista de alertas `machines_alert` da classe.

Recebe quatro parâmetros: `machine_name`, que é uma string representando o nome da máquina; `sensor_type`, que é uma instância do tipo `Datatype` denotando o tipo de sensor que disparou o alerta; `start_time`, uma instância de `datetime` indicando o início do alerta; e `alert_type`, que é uma string representando o tipo de alerta.

Inicialmente, o método cria uma nova instância da classe `MachinesSensorAlert`. Esta

nova instância representa o estado de alerta da máquina. A instância é inicializada com o nome da máquina, o tipo de sensor que causou o alerta, uma marca temporal do início do alerta e o tipo de alerta. Além disso, a máquina é marcada como estando em estado de alerta através do atributo `is_in_alert`, que é definido como `True`.

Finalmente, o novo estado de alerta da máquina, representado pela instância `MachinesSensorAlert` recém-criada, é adicionado à lista `machines_alert`.

```
async def __put_new_machine_in_alert_state(self,
    machine_name:str,
    sensor_type:Datatype,
    start_time:datetime,
    alert_type:str):

    new_machine_alert = MachinesSensorAlert(
        machine=machine_name,
        sensors=[sensor_type.value],
        sensors_historical=[sensor_type.value],
        is_in_alert=True,
        start_time=start_time,
        alert_type=alert_type)

    self.machines_alert.append(new_machine_alert)
```

O método `__remove_machine_sensor_from_alert_state` é uma função assíncrona privada projetada para remover um sensor específico do estado de alerta de uma máquina. Ele recebe três parâmetros: `machine_in_alert`, que é uma instância da classe `MachinesSensorAlert` representando a máquina em questão; `sensor_to_remove`, que é do tipo `Datatype` e identifica o sensor a ser removido; e `end_time`, uma instância de `datetime` que indica o momento em que o sensor foi removido do estado de alerta. Dentro deste método, inicialmente, as posições do sensor e da máquina são identificadas nas listas

apropriadas. O sensor é então removido da lista de sensores em estado de alerta da máquina. Se, após a remoção, a máquina não tiver mais sensores em estado de alerta, ela será removida do estado de alerta, pela chamada do método `__remove_machine_from_alert` caso contrário, apenas o estado do sensor é atualizado, pela chamada de outro método, `__remove_sensor_from_alert_state`.

```
async def __remove_machine_sensor_from_alert_state(self,
    machine_in_alert: MachinesSensorAlert,
    sensor_to_remove:Datatype,
    end_time:datetime):
    index_of_machine = self.machines_alert.index(machine_in_alert)
    index_of_sensor = machine_in_alert.sensors.index(sensor_to_remove.value)

    machine_in_alert.sensors.pop(index_of_sensor)

    if len(machine_in_alert.sensors) == 0:
    await self.__remove_machine_from_alert(index_of_machine, end_time)
    else:
    await self.__remove_sensor_from_alert_state(index_of_machine,machine_in_alert)
```

O método `__remove_machine_from_alert` é outra função assíncrona privada, que tem a responsabilidade de remover completamente uma máquina do estado de alerta. Aceita dois parâmetros: `index_of_machine`, o índice da máquina em questão na lista, e `end_time`, o momento em que a máquina foi removida do alerta. Dentro deste método, a máquina é primeiro marcada como não estando em alerta e depois é removida da lista `machines_alert`. A máquina é então armazenada no banco de dados com um registro de seu estado final e o horário de término. Finalmente, uma notificação é enviada através de um websocket para informar a interface do usuário sobre a mudança no estado da máquina. O detalhamento de como a notificação é enviada está explicada em X

```
async def __remove_machine_from_alert(self,
    index_of_machine:int,
    end_time:datetime):
    machine_in_alert = self.machines_alert[index_of_machine]
    machine_in_alert.is_in_alert = False
    machineNotification = self.machines_alert.pop(index_of_machine)
    machineNotification.end_time = end_time
    await self.iot_database.insert_one(
    NOTIFICATION_DATABASE,
    IOT_MACHINE_ALERTS,
    machineNotification.to_bson())
    await self.websocket.send_notification(machineNotification)
```

O método `__remove_sensor_from_alert_state` é uma função assíncrona simples que atualiza o estado do sensor de uma máquina em alerta na lista de máquinas em alerta. Recebe dois parâmetros: `index_of_machine`, que é o índice da máquina na lista `machines_alert`, e `machine_alert_updated`, que é a instância atualizada da máquina em alerta. Essencialmente, este método substitui a máquina existente na lista pelo objeto atualizado fornecido como parâmetro pelo método `__remove_machine_sensor_from_alert_state`.

```
async def __remove_sensor_from_alert_state(self,
    index_of_machine:int,
    machine_alert_updated:MachinesSensorAlert):
    self.machines_alert[index_of_machine] = machine_alert_updated
```

5.3 Implementação do módulo de processamento de dados

Como explicado em 4.1.2, o modulo de processamento de dados faz a leitura dos dados brutos do sistema, aplica o calculo do `boxplot` e armazena o resultado no banco de dados.

5.3.1 Agendamento para execução periódica

O processamento dos dados precisa ocorrer periodicamente, no caso foi definido inicialmente uma vez por dia. Para executar a chamada da função de processamento uma vez ao dia foi utilizado a biblioteca schedule. Com essa biblioteca foi agendado para todo dia meia noite a execução da função da função de inicia a agregação dos dados. Um loop infinito foi criado para manter o código em execução, verificando se a função deve ser executada ou não.

```
import schedule
schedule.every().day.at("00:00").do(aggregation_init)
print(datetime.now(), flush=True)
while True:
    schedule.run_pending()
    time.sleep(1)
```

5.3.2 Identificando a origem dos dados

Dentro desta estrutura, é necessário identificar as coleções corretas das quais os dados devem ser recuperados antes de realizar o processamento. Essa identificação começa pela função `get_tuples_with_raw_data_collections_and_processed_collections()`. Esta função, como o próprio nome sugere, está encarregada de recuperar tuplas relacionando as coleções de dados brutos com suas respectivas coleções processadas. Itera-se sobre todos os tipos de sensores, representados pelo enumerador `Datatype`, e para cada tipo de sensor, identificam-se as respectivas coleções de dados brutos e processados, resultando em uma lista de tuplas.

Importante destacar que os nomes das coleções são recuperados pelas funções de ajuda, explicados em

```
def get_tuples_with_raw_data_collections_and_processed_collections():
    result:list[tuple] = []
```

```
    for sensor_type in Datatype:
        raw_collection = sensor_name_to_raw_data_collection(
        sensor_type)
        processed_collection = sensor_name_to_processed_collection(
        sensor_type)
        result.append((raw_collection,processed_collection))
    return result
```

Para iniciar o processamento dos dados, a função `aggregation_init()` é chamada, obtendo primeiramente a lista de tuplas que relaciona as coleções de dados brutos com as processadas. Após recuperar esta lista, ela inicializa um loop assíncrono, cujo objetivo é executar uma função de agregação até sua conclusão. Esse design assíncrono é necessário para garantir que o processamento possa realizar chamadas de funções assíncronas, dado que esse modulo é separado da API.

```
def aggregation_init():
    tuples_list =
    get_tuples_with_raw_data_collections_and_processed_collections()

    loop = asyncio.new_event_loop()
    loop.run_until_complete(aggregation(tuples_list))
    loop.close()
```

Desta forma, é identificado a origem dos dados, e onde eles devem ser inseridos depois de processados. Essa informação é passada para a função de agregação para que ela possa ser executada para qualquer dado armazenado.

5.3.3 Iniciando a agregação

Uma vez definida a origem dos dados por meio das coleções identificadas, a fase de agregação dos dados é iniciada. A função responsável por essa tarefa é a `aggregation()`, que aceita uma lista de tuplas representando as coleções de sensores.

```
async def aggregation(sensors_collection_list:list[tuple]):
    database = BaseDB()
    for collection_tuple in sensors_collection_list:
        (raw_data_collection, processed_data_collection) = collection_tuple
        machine_list = await database.read_machines_list(raw_data_collection)

        for machine in machine_list:
            await aggregate_data(
                database,
                raw_data_collection,
                processed_data_collection,
                machine)
```

Dentro desta função, primeiramente, uma instância da base de dados é inicializada usando a classe `BaseDB()`. Em seguida, a função itera sobre cada tupla na lista fornecida. Para cada tupla, as coleções de dados brutos e processados são extraídas. Utilizando a coleção de dados brutos como referência, é feita uma leitura da lista de máquinas associadas a essa coleção por meio do método `read_machines_list()`.

```
async def read_machines_list(self, collection:str):
    temp_client = self.client
    return await temp_client[IOT_DATABASE][collection].distinct('machine')
```

Para cada máquina identificada, os dados são então agregados. A função `aggregate_data()` é chamada, passando-se a base de dados, a coleção de dados brutos, a coleção de dados processados e a máquina específica em questão como argumentos. Esta função, por sua vez, é responsável por realizar a efetiva agregação dos dados da máquina, transformando dados brutos em dados processados que serão armazenados na respectiva coleção de dados processados.

Busca dos dados a serem agregados

Inicialmente, um `query` é gerado utilizando a função `get_aggregation_query()`, que usa as informações da coleção agregada e da máquina em questão. Com esta `query`, os dados brutos são então lidos da coleção de dados brutos usando o método `read_raw_data()`.

A função `get_aggregation_query()` é encarregada de gerar a *query* que busca as informações a serem agregadas pelo modulo de processamento. O objetivo dela é que apenas os dados brutos ainda não processados sejam considerados para agregação, otimizando o processo e evitando reprocessamento desnecessário.

Esta função necessita de uma instância da base de dados, o nome da coleção onde os dados agregados são armazenados e a máquina específica para a qual a agregação é necessária.

```
async def get_aggregation_query(
    database:BaseDB,
    collection:str,
    machine:str)->dict:
    field_to_aggregate = "more_recent_register"
    more_recent_processed_data:BoxPlotData|None =
    await database.read_more_recente_data(
        collection,
        machine,
        field_to_aggregate)

    if more_recent_processed_data is None:
        return __build_query_with_limit_of_data(machine)
    else:
        return __build_query_with_range_of_data(
        more_recent_processed_data,
        machine,
```

```
        field_to_aggregate)
```

A construção da *query* utiliza duas constantes importantes. `MAX_VALUE_BY_PERIOD` armazena a quantidade maxima de registros que podem ser lidos para a agregação, que nesse caso é a quantidade equivalente a 24 horas de leitura considerando a chegada de dados a cada segundo, ou seja 86400 registros. Já `AGGREGATION_PERIOD_IN_HOURS` armazena a quantidade de horas entre uma agregação e outra, no caso 24 horas, sendo condizente com a constante anterior.

Inicialmente, o campo `more_recent_register` é definido como o atributo a ser buscado. A função `read_more_recente_data()` é então chamada para obter os dados processados mais recentes para a máquina e coleção em questão.

```
async def read_more_recente_data(self,
    collection:str,
    machine:str,
    date_time_field:str):
    try:
        temp_client = self.client
        cursor = temp_client[IOT_PROCESSED_DATA][collection].find({"machine":machine}).s
        result:list = await cursor.to_list(None)
        return result[0] if len(result)!= 0 else None
    except Exception as ex:
        print(ex)
        raise ex
```

Se nenhum dado processado recente for encontrado, a *query* é construída utilizando a função `__build_query_with_limit_of_data()`. Esta função simplesmente limita a quantidade de dados recuperados a `MAX_VALUE_BY_PERIOD` e busca por registros que correspondam à máquina especificada.

```
def __build_query_with_limit_of_data(machine:str)->dict:
```

```
    return {
        "limit":MAX_VALUE_BY_PERIOD,
        "query":{"machine":machine}
    }
```

No entanto, se dados processados recentes forem encontrados, a função a ser utilizada é a `__build_query_with_range_of_data()`. Esta função considera o registro processado mais recente e calcula um intervalo de tempo (`date_limit_to_process_data`) adicionando o período de agregação, definido por `AGGREGATION_PERIOD_IN_HOURS`, à data desse registro mais recente. A *query* gerada busca registros com *timestamps* dentro desse intervalo de tempo e que correspondam à máquina especificada, com um limite máximo de registros definido por `MAX_VALUE_BY_PERIOD`.

```
def __build_query_with_range_of_data(more_recent_processed_data:BoxPlotData,
machine:str,
field_to_aggregate:str)->dict:
    date_of_more_recent:datetime =
    more_recent_processed_data[field_to_aggregate]

    date_limit_to_process_data = date_of_more_recent +
    timedelta(hours = AGGREGATION_PERIOD_IN_HOURS)

    return {
        "query":{
            "timestamp": {
                "$gt": date_of_more_recent,
                "$lte": date_limit_to_process_data
            },
            "machine":machine
        },
```

```
        "limit":MAX_VALUE_BY_PERIOD
    }
```

Calculo do BoxPlot

Com query montada, os dados são recuperados com a função `read_raw_data`.

```
async def read_raw_data(self, collection:str, query:dict):
    try:
        temp_client = self.client
        cursor = temp_client[IOT_DATABASE][collection].find(
            query["query"])
            .sort([("timestamp",pymongo.ASCENDING)])
            .limit(query["limit"])
        return await cursor.to_list(None)
    except Exception as ex:
        print(ex)
        raise ex
```

A quantidade de dados recuperados é calculada e, se esta quantidade exceder um valor mínimo predefinido `MINIMUM_DATA_TO_AGGREGATE`, a agregação prossegue. Caso não seja atingindo a quantidade mínima, a função recursiva é finalizada, concluído o processamento dos dados daquela coleção.

`MINIMUM_DATA_TO_AGGREGATE` é um valor constante definido em 100, que garante que existem dados suficientes para serem agregados, evitando a agregação de poucos dados, o que pode comprometer a análise.

Apos a busca da query, um objeto `logger` é inicializado para manter registros do processo de agregação.

Nessa implementação o log é utilizado apenas para mostrar informações no console, mas uma implementação futura pode adicionar uma forma mais completa de logs.

```
class Logger(metaclass=Singleton):
    async def store_aggregation_log(self,
        box_plot_data:BoxPlotData,
        collection:str):
        print("++++++++++++++++++++++++++++++")
        print("Collection {}".format(collection))
        print("more_recent_register {}".format(box_plot_data.more_recent_register))
        print("median {}".format(box_plot_data.median))
        print("mean {}".format(box_plot_data.mean))
        print("q1 {}".format(box_plot_data.q1))
        print("q3 {}".format(box_plot_data.q3))
        print("lower_quartile {}".format(box_plot_data.lower_quartile))
        print("upper_quartile {}".format(box_plot_data.upper_quartile))
        print("mean_with_selection {}".format(box_plot_data.mean_with_selection))
        print("amount_of_data {}".format(box_plot_data.amount_of_data))
        print("++++++++++++++++++++++++++++++")

    async def not_aggregated_data(self, amount:int, collection:str):
        print("==============================")
        print("")
        print("Amount data not aggregated {} - {}".format(str(amount),collection))
        print("==============================")
```

Os dados brutos lidos do banco de dados são convertidos em um DataFrame do pandas (biblioteca da linguagem python utilizada para manipulação e dados), após o qual são calculados os dados agregados relevantes usando a função `calc_box_plot()`. Esta função retorna os dados em uma forma estruturada adequada para representações gráficas, como um box plot.

Para realização do calculo, diversas funções da biblioteca pandas são utilizadas, como

median, quartile, mean e shape, o que facilita o entendimento e a realização do calculo.

```
def calc_box_plot(df:pd.DataFrame, machine:str):
    values = df["value"]

    median = values.median()
    mean = values.mean()

    Q1 = values.quantile(.25)
    Q3 = values.quantile(.75)

    IIQ = Q3 - Q1

    lower_quartile = Q1 - 1.5 * IIQ
    upper_quartile = Q3 + 1.5 * IIQ

    selection = (df["value"]>=lower_quartile) & (df["value"]<=upper_quartile)

    values_selected = values[selection]

    mean_with_selection = values_selected.mean()

    df['timestamp'] = pd.to_datetime(df['timestamp'])
    date_of_more_recent:datetime|str = df['timestamp'].max()

    amount_of_data = df.shape[0]

    box_plot = BoxPlotData()
```

```
box_plot.more_recent_register:datetime = date_of_more_recent

box_plot.lower_quartile=lower_quartile
box_plot.upper_quartile=upper_quartile
box_plot.median=median
box_plot.mean=mean
box_plot.mean_with_selection=mean_with_selection
box_plot.q1=Q1
box_plot.q3=Q3
box_plot.amount_of_data=amount_of_data
box_plot.machine=machine

return box_plot
```

Nessa função, o dataframe recebido contém uma série de valores que será utilizada para calcular os componentes do Box Plot. Primeiro, são determinados os valores da mediana e da média dos dados. Os quartis Q1 (primeiro quartil) e Q3 (terceiro quartil) são calculados utilizando a função `quantile()`, da bilbioteca pandas. A partir destes quartis, o IIQ (intervalo inter quartil) é determinado como a diferença entre Q3 e Q1.

Para identificar os valores discrepantes, são calculados os limites inferior e superior. O limite inferior é obtido subtraindo-se $1.5 \times$ `IIQ` de Q1 e o limite superior é obtido adicionando-se $1.5 \times$ `IIQ` a Q3. Posteriormente, é feita uma seleção dos valores que estão entre os limites inferior e superior. A média destes valores selecionados é então calculada, resultando em `mean_with_selection`.

A função também se encarrega de converter a coluna `timestamp` para o tipo datetime e identificar o *timestamp* mais recente, que será crucial para montagem das buscas nas agregações seguintes.

Com todos os valores calculados, um objeto `BoxPlotData` é instanciado e populado com os componentes do Box Plot, juntamente com informações adicionais, como o número

total de dados e a máquina correspondente.

Registro dos dados processados

Após todo o processo descrito, os dados são convertidos em formato JSON e inseridos na coleção de dados agregados pela função `insert_processed_data`.

```
async def insert_processed_data(self, collection:str, data):
    try:
        temp_client = self.client
        await temp_client[IOT_PROCESSED_DATA][collection]
            .insert_one(data)
    except Exception as ex:
        print(ex)
        raise ex
```

Após a inserção bem-sucedida, a função `aggregate_data()` é chamada recursivamente, garantindo que todos os dados brutos relevantes sejam agregados.

No entanto, se a quantidade de dados brutos não atingir o limite mínimo, a função registra essa ocorrência usando o método `not_aggregated_data()`, indicando que os dados não foram agregados devido à falta deles, e finalização a recursão.

5.4 Implementação da API

Como explicado na seção sobre a arquitetura 4.1.3, a API possui um divisão por módulos, e cada modulo segue uma estrutura pré definida, com uma camada de controller, responsável por receber as requisições HTTP, uma camada de service, responsável por tratar os dados e regras de negócio, e uma camada de respository, responsável por gerenciar o acesso ao banco de dados daquele modulo.

Além disso, a API possui também partes que são comuns a todos os módulos. A infraestrutura que tem a função de disponibilizar uma interface de acesso ao banco de dados,

uma interface de acesso para envio de mensagens web socket, os meios de autenticação, e o gerenciador de threads. Os códigos comuns, que armazenam constantes, funções comuns que precisam ser padronizadas, e modelos de dados.

Portanto, essa seção irá abordar cada uma dessas partes, passando primeiro pelas partes comuns do sistema e depois será mostrado como foi desenvolvido um modulo completo.

5.4.1 Inicialização

A inicialização do sistema acontece por meio do arquivo `main.py`, que serve como o ponto de entrada para inicializar a API e o modulo de processamento de dados. A biblioteca `FastAPI` é utilizada para criar a aplicação principal, aqui referida como `app`. O middleware `CORSMiddleware` é adicionado à aplicação `FastAPI`, permitindo uma configuração de CORS (Cross-Origin Resource Sharing) abrangente. Esta configuração faz com que a API possa ser acessada por diferentes origens.

A importação do módulo `API_data_layer` não apenas incorpora as rotas relacionadas a esse módulo, mas também inicializa o módulo de recebimento de dados. Isso implica que a inicialização deste módulo ocorre simultaneamente ao carregamento da API, porem em uma thread separada, como detalhado em 5.5.1.

Para o gerenciamento de metadados, uma instância do `MetadataRepository` é criada durante a inicialização. Este componente é essencial para o carregamento dos metadados que são utilizados em diferentes partes do sistema, como constantes e parâmetros de alarme.

As rotas da API são então incluídas na aplicação principal através do método `include_router` para diferentes módulos, como autenticação, análise de API, camada de dados, notificações e usuários.

Adicionalmente, o WebSocket é montado na raiz da aplicação através do objeto `socketio_app`, possibilitando a comunicação em tempo real entre o servidor e os clientes. A implementação da conexão websocket pode ser vista em 5.5.1.

A execução do arquivo se conclui com a inicialização do servidor `Uvicorn`, definindo o

host e a porta para escuta. `Uvicorn` é um servidor ASGI (*Asynchronous Server Gateway Interface*) que serve como a interface entre o código da aplicação e o servidor web. Ele é responsável por hospedar a aplicação `FastAPI` e escutar por conexões de entrada no host e na porta especificados. A escolha deste servidor foi pautada na recomendação da documentação do FastAPI.

```
from fastapi import FastAPI
import uvicorn
from fastapi.middleware.cors import CORSMiddleware
from src.infrastructure.database.metadata.metadata_repository import (
MetadataRepository)
from src.modules.api_analytics import api_analytics_router
from src.modules.api_data_layer import api_data_layer_router
from src.modules.notifications import notification_module_router
from src.modules.user import user_module_router, auth_router
from src.infrastructure.websocket import socketio_app
from src.infrastructure.websocket import socket_dispacher
from dotenv import load_dotenv

load_dotenv()
MetadataRepository()
socket_dispacher
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"]
```

```
)
@app.get("/")
async def health_check():
    return {
        "Status":"OK",
        "Message":"Access /docs to more information"
    }
app.include_router(auth_router)
app.include_router(api_analytics_router)
app.include_router(api_data_layer_router)
app.include_router(notification_module_router)
app.include_router(user_module_router)

app.mount("/",socketio_app)

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

5.4.2 Infraestrutura

A infraestrutura e composta de 4 sub-módulos, Autenticação, WebSocket, Conexão com o banco de dados e Gerenciamento de Threads.

Authetication

A implementação da autenticação do sistema foi baseada na documentação oficial do FastAPI, portanto foi adotada uma abordagem baseada em tokens JWT (JSON Web Tokens). JWT é um padrão amplamente aceito para transmitir informações entre partes de maneira segura. A estrutura de um JWT é codificada e pode ser verificada para assegurar que os dados não foram alterados durante a transmissão.

O ponto de entrada para a autenticação é o `o_auth2_password_bearer`, uma instância do OAuth2PasswordBearer que é designada para obter o token a partir do cabeçalho da requisição. O método `auth_middleware` foi definido como um middleware assíncrono, que depende deste bearer token. Esse middleware é utilizado nos controllers para verificar se a requisição recebida tem ou não permissão para acessar as informações.

Dentro deste middleware, a função `decode_jwt_token` é invocada para decodificar e validar o token JWT fornecido.

```
o_auth2_password_bearer = OAuth2PasswordBearer(tokenUrl="/user/login")


async def auth_middleware(token:str = Depends(o_auth2_password_bearer))-> TokenPayload:
    try:
        result = decode_jwt_token(token)
        if result.status:
            return result.data
        else:
            raise HTTPException(status_code=401, detail=result.exception.message)
    except JWSError as jwt_err:
        print(jwt_err)
        raise HTTPException(status_code=401, detail=Unauthorized().message)
```

A função `decode_jwt_token` recebe um token como argumento e tenta decodificá-lo usando a chave secreta e o algoritmo especificados. Se o token for decodificado com sucesso e for do tipo "`access_token`". Caso contrário, diferentes tipos de exceções podem ser levantadas, por exemplo, se o token estiver expirado ou se houver algum erro nas operações de JWT.

```
def decode_jwt_token(token:str)->Result[TokenPayload|None]:
    try:
        token_dict = jwt.decode(token,key=SECRET_KEY, algorithms=ALGORITHM)
```

```
        if token_dict["type"] != "access_token":
            return Result(status=False, exception=WrongTokenType(), data=None)
        token_payload = TokenPayload(**token_dict)
        return Result(status=True, data=token_payload, exception=None)

    except ExpiredSignatureError as invalid_token:
        return Result(status=False, exception=Unauthorized(exception=invalid_token), dat

    except (JWSError, JOSEError, JWTError, JWEError) as ex:
        return Result(status=False, exception=GenericException(message="Authorization er

    except Exception as ex:
        print(ex)
        raise ex
```

A classe modelo para o payload do token é a seguinte:

```
class TokenPayload(BaseModel):
    name:str
    exp:int|None = None
    sub:str|None = None
    user_id:str
    type:str = "access_token"
```

A classe `AuthService` é onde a lógica principal de autenticação é implementada. Esta classe segue o padrão *Singleton* para garantir que apenas uma instância seja criada e usada ao longo da execução do programa.

Dentro de `AuthService`, o método `verify_password` é utilizado para verificar se uma senha fornecida coincide com uma senha criptografada, enquanto o `hash_password` é responsável por criptografar uma senha fornecida.

O método `create_user_tokens` gera um par de tokens (access e refresh) para um usuário, onde o access token é válido por 4 horas e o refresh token por 168 horas. O refresh token é especialmente importante para permitir que os usuários obtenham novos tokens de acesso sem ter que inserir suas credenciais novamente. Se o token de acesso expirar, o token de atualização pode ser usado para obter um novo par de tokens, usando o método `get_new_user_tokens`. Importante destacar que o frontend ainda não faz uso do refresh token, ficando essa funcionalidade para uma futura implementação.

```
class AuthService(metaclass=Singleton):
    def __init__(self):
        self.__database__ = MongoDB()
        self.__user_repository = UserRepository()

        self.__ACCESS_TOKEN_EXPIRE_HOURS__ = 4
        self.__REFRESH_TOKEN_EXPIRE_HOURS__ = 168
        self.__SECRET_KEY__ = SECRET_KEY
        self.__ALGORITHM__ = ALGORITHM
        self.__pwd_context__ = CryptContext(
            schemes=["bcrypt"],
            deprecated="auto")

    def verify_password(self,plain_text_password:str, hashed_password:str):
        return self.__pwd_context__.verify(plain_text_password,hashed_password)

    def hash_password(self,password:str):
        return self.__pwd_context__.hash(password)

    async def create_user_tokens(self, user:User)->tuple[str,str]:
        payload = TokenPayload(name=user.name,user_id=str(user.id))
```

```
        access_token = self.__create_bearer_token(
            user_id=user.id,
            data=payload.__dict__,
            expire_hours=self.__ACCESS_TOKEN_EXPIRE_HOURS__)
        refresh_token = await self.__create_refresh_token(str(user.id))
        return (access_token, refresh_token)

    async def get_new_user_tokens(self,
        refresh_token:str) -> Result[tuple[str, str]]:
        result = self.__decode_jwt_refresh_token(refresh_token)
        if not result.status:
            return Result(status=False, exception=result.exception, data=None)

        token_payload = result.data
        is_valid = await self.__check_if_refresh_token_is_valid(token_payload)
        if not is_valid:
            return Result(status=False, data=None, exception=Unauthorized())

        user = await self.__user_repository.read_user_by_id(token_payload.user)
        payload = TokenPayload(name=user.name,user_id=str(user.id))
        new_access_token = self.__create_bearer_token(
            user_id=token_payload.user,
            data=payload.__dict__,
            expire_hours=self.__ACCESS_TOKEN_EXPIRE_HOURS__)

        return Result(status=True, data=(new_access_token, refresh_token), exception=Non
```

Os métodos `__create_bearer_token` e `__decode_token` são funções auxiliares utilizadas para criar, decodificar e verificar tokens, respectivamente

```
def __create_bearer_token(self,user_id:int, data:dict, expire_hours):
    data_to_enconde = data.copy()
    expire = datetime.now()+timedelta(hours=expire_hours)
    data_to_enconde["exp"] = expire
    data_to_enconde["sub"] = str(user_id)
    return jwt.encode(
        claims=data_to_enconde,
        key=self.__SECRET_KEY__,
        algorithm=self.__ALGORITHM__)

def __decode_token(self,token:str)->dict:
    return jwt.decode(
        token,
        key=self.__SECRET_KEY__,
        algorithms=self.__ALGORITHM__)
```

- Uvicorn usado pelo FastAPI e sua forma asyncrona - Biblioteca Motor usada para acesso ao mongoDB - Biblioteca Pydantic para criação dos modelos e tipos - Web socket para envio de notificações - Biblioteca Jose para autenticação - Comunicação entre as camadas com a classe Result - Contratos de interfaces - Tratamento de erros com classes personalizadas

WebSocket

A implementação da conexão via WebSocket foi feita utilizando a biblioteca socket.io, que possui diversos recursos prontos que facilitam a gestão de conexões Web Socket, com por exemplo a criação de salas para disparo de notificações.

A estrutura adotada para a gestão de conexões Web Socket segue a recomendação de que o cliente deve efetuar uma requisição websocket ao *endpoint* raiz da API para

ser registrado em uma sala virtual específica. Após a conclusão bem-sucedida dessa solicitação, o cliente passa a receber todas as mensagens direcionadas à sala na qual foi cadastrado.

A implementação atual contempla apenas uma sala, destinada especificamente ao envio de notificações relacionadas ao funcionamento das máquinas. Esta sala é identificada pelo identificador `NOTIFICATION_ROOM`. Essa constante armazena o valor `"Notification"`, que é o nome da sala a ser conectada.

O modo assíncrono "asgi"selecionado para a criação do servidor, e as origens permitidas para *CORS* definidas como vazias.

```
socket_io_server = AsyncServer(async_mode="asgi",
    cors_allowed_origins=[])

socketio_app = ASGIApp(socketio_server=socket_io_server,
    socketio_path="")

socket_dispacher = WebSocketDispatcher(socket_io_server)

@socket_io_server.event
async def connect(sid, environ, auth):
    socket_io_server.enter_room(sid, NOTIFICATION_ROOM)
```

O objeto `socket_io_server` é responsável por gerenciar a comunicação *WebSocket*, enquanto o `socketio_app` cria uma aplicação *ASGI* que interage com o servidor *WebSocket*, e é adicionado ao servidor FastAPI, como explicado em **??**. Adicionalmente, uma instância da classe `WebSocketDispatcher` foi criada para facilitar o envio de notificações através do *WebSocket*.

No evento de conexão, denominado `connect`, um cliente é automaticamente adicionado à sala `NOTIFICATION_ROOM`.

Por fim, a classe `WebSocketDispatcher` possui um método `send_notification`, que

é usado para enviar notificações. Ao chamar este método, a notificação é convertida para o formato *JSON* e enviada para todos os clientes na sala `NOTIFICATION_ROOM` através do método `emit`.

```
class WebSocketDispatcher:
    def __init__(self,socket_io_server: AsyncServer):
        self.__socket_io_server = socket_io_server

    async def send_notification(self,
        machine_sensor_notification:MachinesSensorAlert):
        machine_sensors_dict = machine_sensor_notification.to_json()
        await self.__socket_io_server.emit(
            NOTIFICATION_ROOM,
            machine_sensors_dict,
            room=NOTIFICATION_ROOM)
```

A classe `WebSocketDispatcher` é utilizada pelo modulo de recebimento dos dados, detalhado em 5.2.2, para disparar notificações quando é identificado um funcionamento inadequado das maquinas.

Gerenciamento de Threads para Tarefas Assíncronas

Na arquitetura do sistema, foi identificada a necessidade de realizar tarefas de forma concorrente, sem bloquear a execução da API. Essas tarefas são a execução do modulo de recebimento de dados, e a checagem dos metadados do sistema. Ambas tarefas devem ser executadas em paralelo com a execução da API, sem influenciar na sua execução. Portanto, para atingir esse objetivo, um gerenciador de threads, denominado `ThreadManager`, foi implementado.

A classe `ThreadManager` é projetada seguindo o padrão Singleton, assegurando que apenas uma instância seja criada, evitando assim conflitos ou redundâncias no gerenciamento das threads. Uma lista denominada `threads` é inicializada para armazenar todas

as threads criadas, enquanto um loop de eventos assíncronos, atribuído à variável `loop`, é criado utilizando a biblioteca `asyncio`.

O método `start_async_thread` foi introduzido para facilitar a criação e o gerenciamento de tarefas assíncronas. Este método aceita uma função assíncrona, `func`, como argumento e executa as seguintes operações:

1. Uma função interna `start_function` é definida. Esta função é responsável por iniciar a execução da tarefa assíncrona.

2. Dentro de `start_function`, verifica-se a variável booleana `isSeted` para determinar se o loop de eventos já foi configurado. Caso contrário, o loop de eventos é configurado e a tarefa assíncrona é executada até a conclusão através do método `run_until_complete`.

3. Se o loop de eventos já estiver configurado (`isSeted = True`), a tarefa assíncrona é simplesmente adicionada ao loop existente usando `create_task`.

4. Finalmente, uma nova thread é criada com `start_function` como alvo e adicionada à lista `threads`. A thread é então iniciada, executando a tarefa assíncrona.

```
import asyncio
from threading import Thread

class ThreadManager(metaclass=Singleton):
    def __init__(self):
        self.threads = []
        self.loop = asyncio.new_event_loop()
        self.isSeted = False

    def start_async_thread(self,func):
        def start_function():
            if not self.isSeted:
```

```
            asyncio.set_event_loop(self.loop)
            self.isSeted = True
            self.loop.run_until_complete(func())
        else:
            self.loop.create_task(func())

    new_thread = Thread(target = start_function)
    self.threads.append(new_thread)
    new_thread.start()
```

Esta implementação permite a execução de múltiplas tarefas assíncronas em paralelo, cada uma em sua própria thread, todas gerenciadas pelo mesmo loop de eventos assíncronos.

Database

No processo de implementação do sistema, para estabelecer uma conexão eficiente com o banco de dados foi utilizado a biblioteca `motor` foi adotada como mecanismo.

No centro da estratégia de conexão está uma classe base, denominada `BaseDB`, que tem a responsabilidade não apenas de estabelecer a conexão com o MongoDB, mas também de definir uma série de operações básicas para a manipulação dos dados armazenados. A estrutura dessa classe é apresentada a seguir:

```
import motor
class BaseDB:
    def __init__(self):
        self.client = motor.motor_tornado.MotorClient(url, port)
```

Algumas das operações fundamentais implementadas por `BaseDB` incluem:

- `insert_one`: Recebe como parâmetros o *database* e a *collection* correspondentes em formato de texto, e a *data* a ser inserida. Insere um documento na coleção especificada.

- `insert_many`: Recebe como parâmetros o *database* e a *collection* correspondentes em formato de texto, e a *data* contendo vários documentos a serem inseridos. Insere vários documentos na coleção especificada.

- `read_data_with_pagination`: Recebe como parâmetros o *database*, a *collection*, a *query*, o *page_number*, o *limit*, o *sort_descending_field* e a *projection*. Recupera dados com paginação, permitindo uma leitura mais organizada.

- `read_data_with_limit`: Recebe como parâmetros o *database*, a *collection*, a *query* e o *limit*. Lê dados com um limite predefinido de documentos retornados.

- `read_data`: Recebe como parâmetros o *database*, a *collection* e a *query*. Realiza uma leitura simples de dados baseada em uma query.

- `get_distinct_property`: Recebe como parâmetros o *database*, a *collection* e a *property*. Obtém propriedades distintas de uma coleção, verificando todos os documentos presentes.

- `list_collections_by_db`: Recebe como parâmetro o *database*. Lista todas as coleções presentes em um banco de dados específico.

- `add_item_into_lists_by_filter`: Recebe como parâmetros o *database*, a *collection*, o *filter*, as *list_properties* e a *new_data*. Adiciona um item em listas específicas baseado em um filtro.

- `update_item`: Recebe como parâmetros o *database*, a *collection*, a *data* a ser atualizada e o *filter*. Atualiza um documento específico.

- `update_many_items`: Recebe como parâmetros o *database*, a *collection*, a *data* a ser atualizada e o *filter*. Atualiza vários documentos que atendam a um filtro.

- `count_documents`: Recebe como parâmetros o *database*, a *collection* e a *query*. Conta o número de documentos em uma coleção que atendem a uma consulta.

- `get_data_between_dates`: Recebe como parâmetros o *database*, a *collection* e a *query*. Recupera dados entre duas datas específicas.

Com a base de acesso estabelecida, outras classes foram desenvolvidas, herdados de `BaseDB`, para atender contextos específicos do sistema. Essas classes seguem o padrão singleton, o que garante que apenas uma instância da conexão seja criada para um contexto específico, otimizando a gestão dos recursos. Um exemplo é a classe `MongoDBIOT` destinada ao módulo de recebimento de dados:

```
class MongoDBIOT(BaseDB, metaclass=Singleton):
    def __init__(self):
        super().__init__()
```

Classes semelhantes, seguindo o mesmo formato, foram criadas para outros contextos, como o acesso ao banco de dados pela API, garantindo uma estrutura organizada e eficiente de conexão e manipulação dos dados.

5.4.3 Arquivos comuns

Dentro da estrutura da API, uma pasta denominada `common` foi implementada com o intuito de centralizar componentes reutilizáveis, abrangendo múltiplos módulos e camadas. Esta organização foi estabelecida para maximizar a eficiência do desenvolvimento e a consistência do código.

Modelos de Dados

A seção de modelos de dados na pasta `common` abriga diversas classes que definem a estrutura dos dados utilizados. Classes que especificam usuários e dados de sensores estão presentes. Para exemplificar temos a classeNotificationSensorResponse, que é responsável por definir o modelo de dados que é retornado quando a API recebe uma requisição solicitando as notificações de um determinado usuário. Destaca-se o uso do `BaseModel` na sistema de herança, do `Pydantic`, necessário para definir os tipos de retorno dentro do

FastAPI. Além disso é utilizado a função `Field`, também do `Pydantic`, para indicar com o três pontos que é um atributo obrigatório de ser informado na construção da classe.

```
class NotificationSensorResponse(BaseModel):
    data:list[dict] = Field(...)
    total_count:int = Field(...)
```

Outros modelops de dados importantes são as classes de exceções e uma classe denominada `Result`, responsável pelo tráfego de dados entre as diferentes camadas da aplicação, mostrado na implementação do modulo da API em X.Exemplo de classe que define uma exceção do sistema.

```
class CustomBaseException(Exception):
    def __init__(self, message:str, exception, *args: object) -> None:
        self.message = message,
        self.exception = exception
        super().__init__(*args)


class GenericException(CustomBaseException):
    def __init__(self,
        message:str =   "An error has occurred",
        exception = None) -> None:


        super().__init__(message, exception)
```

Classe `Result` usado para comunicação entre camadas dos módulos do sistema. O `TypeVar` é utilizado para indicar um tipo genérico para o atributo `data` da classe.

```
from typing import TypeVar, Generic
from src.common.models.exceptions.unauthorized import CustomBaseException


T = TypeVar('T')
```

```
class Result(Generic[T]):
    def __init__(self, status:bool, data:T|None, exception:CustomBaseException|None):
        self.status = status
        self.data = data
        self.exception = exception
```

Destaca-se também o uso da classe `Singleton`, utilizada quando a existe a necessidade de garantir que apenas umas instância de determina classe será utilizada.

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]
```

Por último, é destacado a classe `PyObjectId`, utilizada para definir um tipo para o atributo ID das classes que representam modelos do banco de dados. Os métodos definidos para essa classe são usados internamente pelo `FastAPi` e pelo `Pydantic`.

Essa classe é necessária para que o ID possa ser corretamente convertido para texto, e retornado nas requisições. Além disso, o uso dessa classe viabiliza a manipulação do ID quando necessário, deixando a gestão dos identificadores únicos para a aplicação e não para o banco de dados, como uma boa prática para desenvolvimento de sistemas.

```
class PyObjectId(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate


    @classmethod
    def validate(cls, v):
```

```
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid object id")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")
```

Funções Helpers

A seção de funções *helpers* na pasta `common` foi construída para conter métodos que sejam comum em diferentes módulos do sistema, evitando a duplicação de código e padronizando o funcionamento.

Um exemplo dessas funções é a conversão de `Datatype` para coleções do MongoDB. O código a seguir ilustra este processo:

```
def sensor_name_to_processed_collection(
    sensor_name:Datatype)->str:

    sensor = (sensor_name.name).capitalize()
    return IOT_AGGREGATION_COLLECTION.replace("NAME", sensor)

... (outras funções relacionadas)
```

Estas funções são empregadas para determinar os nomes corretos das coleções em de acordo com o tipo de dado a ser tratado e manipulado pelos modulos. Funções como `sensor_name_to_processed_collection` e `sensor_name_to_raw_data_collection` convertem entre nomes de sensores e nomes de coleções. Adicionalmente, uma série de funções foi desenvolvida para mapear o nome processado da coleção, de volta para o `Datatype` correspondente, garantindo uma manipulação de dados mais segura e coerente.

Constantes

Por fim, a seção de constantes armazena uma série de valores fixos que são usados em várias partes do sistema. Isso inclui nomes de bancos de dados, tipos de alertas, salas de websocket, e outros que forem necessários.

Dentre as constantes destaca-se a `DataType`, que é um *enum* que padroniza os tipos de dados que podem ser recebidos dos sensores. Esta padronização é empregada em diversos módulos para assegurar que os dados sejam recebidos, processados, armazenados e retornados de maneira consistente e correta.

```
from enum import Enum


class Datatype(Enum):
    PRESSURE = "PRESSURE"
    TEMPERATURE = "TEMPERATURE"
    VOLTAGE = "VOLTAGE"
    CURRENT = "CURRENT"
    SPEED = "SPEED"
    ACCELERATION = "ACCELERATION"
    DISTANCE = "DISTANCE"
    HUMIDITY = "HUMIDITY"
    FORCE = "FORCE"
    PRODUCTION_COUNTER = "PRODUCTION_COUNTER"
```

5.4.4 Módulos

A API foi estruturada em módulos, cada um com responsabilidade para gerenciar um determinado contexto. Abaixo são listados os módulos desenvolvidos 5.4.2.

1. **IOT Analytics**: Modulo responsável por gerenciar o acesso as informações dos sensores, tanto os dados em tempo real via stream, quanto as informações processadas pelo modulo de processamento de dados, explicado em 5.3.

2. **Notifications**: Modulo responsável por gerenciar o acesso as notificações.

3. **User**: Modulo responsável por gerir as informações dos usuários do sistema, e realização da autenticação explicada em 5.4.2.

4. **Downtime Analytics**: Modulo usado para disponibilizar os dados de teste da paragem das maquinas. Esse modulo alimenta a tela que exibe as informações de paragem das maquinas.

Cada modulo seguiu um padrão de ter uma camada para o recebimento das requisições HTTP, o `controller`, outra para tratar a requisição de acordo com as regras de negócio, o `service`, e o `repositoy`, para disponibilizar métodos de acesso e manipulação das informações no banco de dados.

Controller

A implementação dos módulos da API passa pela construção de três camadas essenciais. O controller para receber as requisições HTTP, enviar as informações recebidas para a camada de serviço, e realizar o retorno adequado, com as informações formatadas, e código HTTP correto.

Para exemplificar o funcionamento do controller, um exemplo específico será apresentado. O seguinte fragmento de código representa o router da API para os dados dos sensores IoT (*Internet of Things*):

```
iot_data_router = APIRouter(tags=["IOT Data"], dependencies=[Depends(auth_middleware)])


service = ServiceIOT()


@iot_data_router.get("/realtime")
async def real_time_iot():
    def get_real_time_data():
        sensor = SensorValue()
```

```
        while True:
            time.sleep(1)
            lista_json = [machine.to_json() for machine in sensor.machine_list]
            last_data = json.dumps(lista_json)
            yield bytes(last_data, "utf-8")

    return StreamingResponse(
        get_real_time_data(),
        media_type="application/octet-stream")
```

Neste exemplo, o controller faz uso do `APIRouter` para criar rotas associadas aos dados da IoT. Um middleware de autenticação é aplicado como uma dependência, como mostrado no detalhamento da autenticação em ref X, garantindo que apenas usuários autenticados possam acessar essas rotas. Nesse caso, ao aplicar o middleware na criação do `iot_data_router`, é garantido que todos os `endpoints` criados a partir dele precisem de autenticação.

Logo abaixo é criada uma instância do serviço que deve ser utilizado pelos `endpoints` para responder as requisições recebidas.

A rota `/realtime` é designada para fornecer dados em tempo real. Uma função interna, `get_real_time_data`, é definida e responsável por coletar esses dados do `SensorValue`, explicado em ref X. Nesse endpoint, a API pega os valores atuais e retorna uma `StreamingResponse`, que envia os dados em tempo real como um fluxo contínuo.

As outras rotas, como `/machines_in_sensor` e `/graph_info`, operam de maneira semelhante, mas com diferentes responsabilidades. Elas fazem chamadas para o `ApiRepository` para recuperar informações específicas e retorná-las ao cliente. Caso ocorra uma exceção ou erro, um `HTTPException` é lançado com um código de status HTTP apropriado e uma mensagem de erro detalhada.

Nesses dois endpoints é importante destacar a especificação do `response_model`, para que ocorra validações automáticas pelo framework antes do dado ser retornado, garantido

a consistência nos dados retornados.

```
@iot_data_router.get("/machines_in_sensor", response_model=list[MachinesSensor])
async def get_machines_in_sensor():
    result = await service.get_all_machines_processed_info()
    if result.status:
        return result.data

    raise HTTPException(status_code=500, detail=result.exception.message)

@iot_data_router.get("/graph_info", response_model=list[ProcessedData])
async def get_graph_info(machine:str, sensor:Datatype, initial_date:datetime, end_date:d
    result = await service.get_processed_data(
        machine=machine,
        data_type=sensor,
        initial_date=initial_date,
        end_date=end_date)

    if result.status:
        return result.data

    raise HTTPException(status_code=500, detail=result.exception.message)
```

Service

A camada de serviço serve para aplicar as devidas regras antes de retornar os dados para a camada de controle 5.4.4. Pode acessar a camada de repositório para realizar a leitura ou escrita de dados, e deve retornar os dados para a camada de controle usando a classe `Result`, mostrada na seção 5.4.3, para que possa ser identificado corretamente o resultado do processamento realizado, e o retorno adequado seja dados para o cliente que realizou

a requisição.

Para uma compreensão mais profunda, um exemplo específico da implementação da camada de serviço segue abaixo:

```
class ServiceIOT:
    def __init__(self):
        self.__repository = RepositoryIOT()
        self.__database__ = MongoDB()
        self.appMetadata = MetadataRepository()

    async def get_processed_data(self,
        machine:str,
        data_type:Datatype,
        initial_date:datetime,
        end_date:datetime)-> Result[list[ProcessedData]]:
        try:
            alert_parameter = await self.appMetadata.get_sensor_alert_value(
            data_type)

            processed_data = await self.__repository.read_iot_processed_data__(
                machine=machine,
                datatype=data_type,
                initial_date=initial_date,
                end_date=end_date,
                sort_by_field="more_recent_register")

            for data in processed_data:
                data["alert_parameter"] = alert_parameter
            return Result[list[MachinesSensor]](
```

```
                status=True,
                data=processed_data,
                exception=None)
        except Exception as ex:
            return Result[list[MachinesSensor]](
                status=False,
                data=None,
                exception=GenericException(exception=ex))
```

Nesta implementação, a classe `ServiceIOT` é inicializada com instâncias de `RepositoryIOT` e `MetadataRepository`, permitindo que o serviço acesse as camadas de repositório e metadados correspondentes.

O método `get_processed_data` serve para obter dados processados com base em diversos parâmetros como máquina, tipo de dados e intervalo de datas. Inicialmente, um valor de alerta é recuperado do metadado do sensor para o tipo de dados fornecido. Posteriormente, os dados processados são lidos do repositório. A cada entrada de dados recuperada, o valor de alerta é adicionado como um novo campo. O método retorna um objeto `Result` encapsulando esses dados. O objeto `Result` é detalhado em X.

O método `get_all_machines_processed_info` recupera informações sobre todas as máquinas e sensores processados. Ele itera através das coleções de dados processados, agregando informações de máquinas e sensores. Em caso de sucesso, ele retorna um objeto `Result` contendo uma lista de máquinas e seus sensores correspondentes.

```
async def get_all_machines_processed_info(
    self)-> Result[list[MachinesSensor]]:
    try:
        collection_list =
            await self.__repository.get_processed_data_collections()
        machine_sensors: list[MachinesSensor] = []
        for collection in collection_list:
```

```
            machine_list = await self.__repository.get_distinct_machines_by_collection(c
            sensor = processed_collection_to_sensor_name(collection)

            for machine in machine_list:
                matching_sensor =
                next((data for data in machine_sensors
                    if data.machine == machine), None)
                if matching_sensor:
                    matching_sensor.sensors.append(sensor)
                else:
                    machine_sensors.append(MachinesSensor(
                        machine=machine,
                        sensors=[sensor]))

        return Result[list[MachinesSensor]](
            status=True,
            data=machine_sensors,
            exception=None)
    except Exception as ex:
        print(ex)
        return Result[bool](
            status=False,
            data=None,
            exception=GenericException(exception=ex))
```

O último método, `read_raw_data_by_id`, serve para ler dados brutos com base em um identificador e tipo de dados. Ele acessa o repositório correspondente para recuperar os dados e retorna um objeto `Result` contendo esses dados ou uma exceção, se aplicável.

```
async def read_raw_data_by_id(self,
```

```
    raw_data_id:str,
    datatype:Datatype) -> Result:
    try:
        collection = sensor_name_to_raw_data_collection(datatype)
        result = await self.__repository.read_raw_data(collection,raw_data_id)
        return Result(status=True, data=result, exception=None)
    except Exception as ex:
        return Result[bool](
            status=False,
            data=None,
            exception=GenericException(exception=ex))
```

Esta implementação exemplifica como a camada de serviço interage com as camadas de repositório e acessa metadados metadados, e como ela prepara os dados para serem enviados de volta ao controller, garantindo assim um fluxo de dados coeso e eficiente através das diversas camadas da aplicação.

Respository

Por fim a camada a camada de repositório é responsável por ter acesso ao banco de dados e realizar as operações de leitura e escrita de acordo com as necessidades. Essa camada instancia uma conexão com banco de dados, e seus métodos utilizam os métodos base definidos pela infraestrutura de conexão com o banco, em 5.4.2, para realizar a operações de acordo com o contexto daquele modulo.

O código a seguir fornece um exemplo da implementação dessa camada:

```
class RepositoryIOT:
    def __init__(self):
        self.__database__ = MongoDB()
```

```
async def get_processed_data_collections(self):
    collection_list =
        await self.__database__.list_collections_by_db(
        IOT_PROCESSED_DATA)

    return collection_list

async def get_distinct_machines_by_collection(self, collection:str):
    machine_list =
        await self.__database__.get_distinct_property(
            IOT_PROCESSED_DATA,
            collection,
            "machine")

    return machine_list

async def read_raw_data(self, collection:str, raw_data_id:str):
    result = await self.__database__.read_data(
        IOT_DATABASE,
        collection,
        {"_id":ObjectId(raw_data_id)})

    return result

async def read_iot_processed_data__(self, machine:str, datatype:Datatype,
    initial_date:datetime, end_date:datetime,
    sort_by_field:str) -> list[ProcessedData]:

    try:
```

```
        collection = sensor_name_to_processed_collection(datatype)
        query = {
            "machine":machine,
            sort_by_field:{
                "$gte":initial_date,
                "$lte":end_date
            }
        }
        result = await self.__database__.get_data_between_dates(
            IOT_PROCESSED_DATA,
            collection,query)
        return result
    except Exception as ex:
        print(ex)
        raise ex
```

Na inicialização da classe `RepositoryIOT`, uma conexão com o MongoDB é instanciada.

O método `get_processed_data_collections` é utilizado para listar todas as coleções de dados processados do banco de dados `IOT_PROCESSED_DATA`. Este método faz uma chamada direta ao método de listagem de coleções fornecido pela classe `MongoDB`.

O método `get_distinct_machines_by_collection` é responsável por recuperar uma lista de máquinas distintas para uma determinada coleção. Ele faz isso através do método `get_distinct_property` do banco de dados.

O método `read_raw_data` serve para ler dados brutos de uma coleção específica, utilizando o ID dos dados como parâmetro de pesquisa. Ele faz uma chamada ao método `read_data` da classe `MongoDB`, fornecendo os parâmetros necessários para a leitura dos dados.

Por fim, o método `read_iot_processed_data__` é utilizado para ler dados processados

com base em diversos critérios como máquina, tipo de dados e intervalo de datas. Uma query é construída para esse fim e passada ao método `get_data_between_dates` da classe `MongoDB`.

Cada um desses métodos auxilia na manutenção de uma separação clara de responsabilidades, permitindo que a camada de serviço mantenha um foco estrito na lógica de negócios, enquanto a camada de repositório gerencia as operações do banco de dados.

5.5 Implementação do frontend

A implementação da interface de usuário, foi desenvolvida de acordo com a arquitetura exposta na seção X. O desenvolvimento do *frontend* é segmentado em várias partes, que incluem a organização das páginas do sistema conforme a estrutura pré definida do Next.js, o gerenciamento de dados acessados pelos componentes, a configuração para acesso externo, a manipulação e o acesso às imagens (*assets*), e a construção dos componentes individuais.

É relevante notar que, para manter a conformidade com as melhores práticas e simplificar o desenvolvimento, as configurações padrão do Next.js foram mantidas.

5.5.1 Paginas do sistema

Por se tratar de um framework, o NextJs tem uma estrutura pré definida para criação das páginas do sistema assim como suas rotas. Dentro dos arquivos do framework, a pasta *pages* é utilizada para armazenar cada uma das páginas do sistema, sendo cada arquivo uma página, e o nome do arquivo sendo a rota para acesso. A configuração das páginas acorre por arquivos com nomes específicos, no caso *_app.tsx* e *_document.tsx*.

Paginas de configuração

Em relação as páginas de configuração temos primeiro a *_app.tsx*. Esse arquivo tem a responsabilidade de configurar e gerenciar contextos, estilização global e a localização de

datas, ou seja, aspectos globais de toda a aplicação.

O código começa pela importação de diversos módulos e bibliotecas, o que inclui contextos específicos como `OpenContext` e `PrivateContext`, que são explicados melhor em **??**, e o suporte para localização de datas com `LocalizationProvider` e `AdapterDayjs`.

O `LocalizationProvider` e `AdapterDayjs` são de bibliotecas que têm o objetivo de fornecer funcionalidades de localização e formatação de datas. O `LocalizationProvider` atua como um encapsulador para o sistema de datas, permitindo a integração com diferentes bibliotecas de gerenciamento de datas. Neste caso, o `AdapterDayjs` é utilizado como o adaptador para a biblioteca Day.js, permitindo que as datas sejam manipuladas e formatadas de maneira eficiente e compatível com diversos locais geográficos e formatos. Com essas bibliotecas fica mais fácil gerenciar datas para a construção dos filtros do dashboard, explicados em X.

O tipo `NextPageWithLayout` foi definido para enriquecer as propriedades da página com informações sobre o *layout*. Isso permite que cada página tenha um *layout* personalizado se necessário, oferecendo grande flexibilidade no design da interface.

A função principal `App`, que recebe `Component` e `pageProps` como argumentos, é responsável por configurar o *layout* e renderizar os componentes da página. A lógica dentro desta função verifica a rota atual usando `useRouter` para determinar se o usuário está na página de login.

O conteúdo é então encapsulado dentro dos contextos relevantes. Se o usuário estiver na página de login, apenas o `OpenContext` é aplicado. Para todas as outras páginas, o `PrivateContext` é adicionalmente aplicado, garantindo que as informações sensíveis sejam acessadas apenas por usuários autenticados. Os contextos utilizados são detalhados em ref.

Dentro do `LocalizationProvider`, o adaptador `AdapterDayjs` é utilizado para fornecer funcionalidades de localização de datas, tornando o aplicativo mais versátil em diferentes locais.

```
import {OpenContext, PrivateContext} from '@/context'
```

```
import '@/styles/globals.css'
import { LocalizationProvider } from '@mui/x-date-pickers'
import { AdapterDayjs } from '@mui/x-date-pickers/AdapterDayjs'
import { NextPage } from 'next'
import type { AppProps } from 'next/app'
import { useRouter } from 'next/router'
import { ReactElement, ReactNode } from 'react'

export type NextPageWithLayout<P = {}, IP = P> = NextPage<P, IP> & {
    getLayout?: (page: ReactElement) => ReactNode
}

type AppPropsWithLayout = AppProps & {
    Component: NextPageWithLayout
}

export default function App({ Component, pageProps }:
    AppPropsWithLayout) {

    const getLayout = Component.getLayout || ((page) => page)
    const router = useRouter()
    const isLoginPage = router.pathname === "/"

    const componentWithProps = <Component {...pageProps} />

    return getLayout(
    <LocalizationProvider dateAdapter={AdapterDayjs}>
        <OpenContext>
```

```
        {isLoginPage?
            <>{componentWithProps}</>
            :<PrivateContext>
                {componentWithProps}
            </PrivateContext>
        }

        </OpenContext>
    </LocalizationProvider>
    )
}
```

Embora seja um arquivo mais simples comparado ao `_app.tsx`, o `_document.tsx` tem a responsabilidade de definir da estrutura HTML global da aplicação.

No arquivo, foram importados os componentes `Html`, `Head`, `Main`, e `NextScript` da biblioteca `next/document`. Estes componentes são utilizados para criar a estrutura básica da página HTML dentro do NextJs.

O componente `Html` é utilizado para encapsular todo o conteúdo HTML e inclui o atributo `lang="en"`, o qual define o idioma da página como inglês. O componente `Head` é empregado para adicionar elementos no cabeçalho da página HTML. Neste caso, o título da página é definido como 'Dashboard'.

O corpo da página HTML é composto pelos componentes `Main` e `NextScript`. O `Main` é o local onde o conteúdo principal da página é inserido, enquanto o `NextScript` é responsável por incluir os scripts necessários para o funcionamento do Next.js.

Vale destacar que o `_document.tsx` não tem acesso a características específicas da página como os métodos `getInitialProps`, `getStaticProps`, ou `getServerSideProps` (funções do NextJs para carregado de dados do lado do servidor). Isso implica que este arquivo é ideal para configurações que são comuns em todas as páginas e não requerem informações dinâmicas.

```
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html lang="en">
      <Head title='Dashboard'/>
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

Paginas do sistema

As páginas do sistema se dividem em dois tipos, privadas e publica, sendo que publica é apenas a página de login. Essa página pública está no arquivo `index.tsx`, sendo a rota raiz do sistema.

Neste arquivo, apenas configurações `meta` e o componente `Login` são invocados. O elemento `Head` é utilizado para definir configurações globais do HTML, como o título da página e metadados.

O componente `Login` é chamado dentro da tag `main`, que serve como o conteúdo principal da página. Esta abordagem de design mantém a página `index.tsx` enxuta, transferindo a maior parte da lógica e da apresentação visual para o componente `Login`. Este é um exemplo do princípio de separação de interesses, onde cada arquivo ou componente tem uma única responsabilidade claramente definida.

```
export default function Home() {
    return (
```

```
    <>
      <Head>
        <title>Catraport Dashboard</title>
        <meta name="description"
          content="Generated by create next app" />
        <meta name="viewport"
          content="width=device-width,
          initial-scale=1" />
        <link rel="icon" href="/favicon.ico" />
      </Head>
      <main>
        <Login/>
      </main>
    </>
  )
}
```

As outras páginas do sistema se encontram dentro da pasta `dashboard`, que também está dentro da pasta `pages`. Isso implica que todas as páginas dentro dessa pasta devem ser acessados na rota `/dashboard`.

Dentro do dashboard, existe a páginas principal, em `/index.tsx`, com a página do dashboard que exibe os dados em tempo real e os gráficos com os dados históricos processados. As funcionalidades dessa página está detalhada em X.

Este arquivo segue à mesma lógica de design observada na página de login, mantendo a separação entre as configurações da página e a lógica dos componentes invocados.

O componente `Dashboard` se baseia em composição, delegando diversas responsabilidades a componentes individuais. O componente `DashboardLayout` é utilizado como um contêiner que define a estrutura global da página, oferecendo um layout consistente também para as outras páginas do dashboard. Dentro deste componente, vários outros

são chamados para realizar funções específicas.

O `DashboardHeader` é responsável pela exibição do cabeçalho da página, fornecendo o acesso aos filtros para visualização das informações. Segue-se o componente `SensorsValues`, que é designado para mostrar os valores dos sensores em tempo real.

Um elemento `Divider`, da biblioteca `Material UI 5` é inserido para fornecer uma separação visual entre as diferentes seções da página. Por fim, o componente `SensorsGraphs` é invocado para exibir gráficos relacionados aos dados históricos dos sensores de forma agregada.

```
export default function Dashboard() {
    return (
        <DashboardLayout>
            <DashboardHeader/>
            <SensorsValues/>
            <Divider/>
            <SensorsGraphs/>
        </DashboardLayout>
    )
}
```

As outras páginas do dashboard também foram construídas usando a lógica de composição demostrada e utilizando o mesmo componente base para o layout, `DashboardLayout`. Essas páginas são:

1. **Maintenace**: Responsável por exibir os dados de paragem das maquinas em forma de gráficos para demostrar a visualização dessas informações dentro do sistema.

2. **Profile**: Responsável por exibir as informações do usuário que está logado no sistema, assim como permitir realizar alterações nos dados.

5.5.2 Gerencia dos dados do sistema

Um dos pontos importantes no desenvolvimento do frontend é o gerenciamento de estados globais, que são informações ou comportamentos compartilhados entre componentes não relacionados. Dentro desse projeto, a Context API foi utilizada para este propósito, senso uma solução nativa do React que se destaca pela sua facilidade de implementação e utilização. Esse recurso permite que dados sejam passados de forma eficiente em toda a árvore de componentes, eliminando a necessidade de passar manualmente propriedades através de níveis intermediários.

Não apenas estados de dados, mas também estados comportamentais, como o estado de login do usuário e o estado do menu lateral (aberto ou fechado), foram gerenciados por meio do Context API. A camada de dados foi construída de tal forma que todas as informações necessárias para o funcionamento do sistema, que dependem de um valor global, foram incluídas.

Os contextos criados para o gerenciamento de estados são os seguintes:

1. `SnackbarContext`: Utilizado para o gerenciamento de mensagens e alertas no sistema, facilitando o acesso a função que exibe alertas e sua configuração em todo o sistema.

2. `AuthContext`: Encarregado de gerenciar o estado de autenticação do usuário.

3. `ThemeContext`: Responsável pela gestão do tema visual da aplicação. ref para o MUI5

4. `NotificationContext`: Utilizado para o gerenciamento, leitura e recebimento de notificações no sistema.

5. `LegacyContext`: Gerencia informações de parada das máquinas, após serem lidas do backend.

6. `IotContext`: Utilizado para o gerenciamento de estados relacionados aos dispositivos IoT no sistema.

7. `DrawerContext`: Encarregado de gerenciar o estado do menu lateral (aberto ou fechado), e disponibilizar em todas as páginas do sistema que utilizam o menu lateral.

Cada contexto foi concebido com uma função específica, de modo a permitir uma separação clara das responsabilidades. Utilizando a arquitetura mostra em X, esse modelo de gerenciamento de dados no frontend fica simples e escalável, como especificado no requisitos.

Gerencia de diferentes contextos

Para tratar da complexidade gerada pela variedade de contextos necessários no sistema, optou-se por instanciar esses contextos por meio de componentes específicos. Esses componentes foram projetados para encapsular diferentes grupos de contextos, de acordo com as necessidades de acesso.

Dois componentes principais foram desenvolvidos: `OpenContext` e `PrivateContext`. O primeiro é responsável por instanciar os contextos que estão disponíveis para qualquer indivíduo que acessar o sistema. O segundo é encarregado de instanciar contextos que só podem ser acessados por usuários autenticados. A criação de um contexto é exemplificado melhor em 5.5.2.

Esses componentes são usados no arquivo `__app.tsx`, que é o arquivo de configuração inicial do Next.js, como explicado em 5.5.1, de modo a tornar os contextos acessíveis em toda a árvore de componentes da aplicação.

A seguir, é apresentado o código que ilustra como esses componentes foram implementados:

```
interface Props{
    children:React.ReactNode
}

function OpenContext({children}:Props){
```

```
    return (
        <SnackbarContextProvider>
            <AuthContextProvider>
                <ThemeContextProvider>
                        {children}
                </ThemeContextProvider>
            </AuthContextProvider>
        </SnackbarContextProvider>
    )
}

function PrivateContext({children}:Props){
    return(
        <>
            <NotificationProvider>
                <LegacyContext>
                    <IotContext>
                        <DrawerContextProvider>
                            {children}
                        </DrawerContextProvider>
                    </IotContext>
                </LegacyContext>
            </NotificationProvider>
        </>
    )
}
```

Desta forma, os contextos são adequadamente isolados e gerenciados, garantindo que os dados e funcionalidades corretos estejam disponíveis para os usuários, de acordo com

seu nível de acesso.

Criação de um contexto

Para a criação de um contexto, primeiro é necessário a definição correta dos tipos, mandatório devido ao uso do `Typescript`. Para cada contexto, uma interface de propriedades (`Props`) e um valor padrão (`DEFAULT_VALUE`) são criados.

Para mostrar a criação de um contexto dentro do projeto será usado como exemplo o contexto `DrawerContext`, responsável por gerenciar o estado do drawer da aplicação. O código a seguir exemplifica como este contexto foi criado:

```
import { createContext, useContext, useState } from "react"

interface Props {
    open:boolean
    setOpen:React.Dispatch<React.SetStateAction<boolean>>
}

const DEFAULT_VALUE = {
    open:false,
    setOpen:()=>{}
}

const DrawerContext = createContext<Props>(DEFAULT_VALUE)

function DrawerContextProvider({ children }:{children:React.ReactNode}){
    const [open, setOpen] = useState<boolean>(false)

    return (
        <DrawerContext.Provider value={{open,setOpen}}>
```

```
            {children}
        </DrawerContext.Provider>
    )
}

export default function useDrawer(){
    return useContext(DrawerContext)
}

export {DrawerContextProvider};
```

Neste exemplo, o contexto `DrawerContext` é criado utilizando a função `createContext` do React. A interface `Props` define os tipos para o estado aberto do drawer (`open`) e a função para definir esse estado (`setOpen`). Um valor padrão (`DEFAULT_VALUE`) é estabelecido para inicializar o contexto.

O componente `DrawerContextProvider` utiliza o estado React local para gerenciar o valor do estado `open`. Este valor e a função `setOpen` são então disponibilizados para todos os componentes filhos por meio do `DrawerContext.Provider`. Dessa forma, o componente `DrawerContextProvider` é usado no gerenciamento dos contexto, como explicado em 5.5.2, para tornar acessível toda a informação para a árvore de componentes inteira.

A função `useDrawer` é uma função personalizada que facilita o acesso ao contexto `DrawerContext` em qualquer parte da aplicação. Dentro do react, funções com o 'use' na frente são denominadas `hooks`, como pode ser visto na documentação oficial em X.

Dessa forma, o contexto foi criado e pode ser utilizado para gerenciar o estado de aberto e fechado do menu lateral em toda a aplicação.

Esse modelo de criação de criação de contextos se repete para todos os outros contextos, com a adição do acesso ao backend, que é explicado em 5.5.3.

5.5.3 Acesso externo

Para a interação com dados armazenados e gerenciados pelo backend, foi estabelecida uma camada de acesso externo no frontend. Essa camada serve como um ponto centralizado para todas as requisições de rede e é necessário para a leitura e manipulação de dados que estão fora do escopo do frontend, portanto lida com as requisições HTTP, conexão WebSocket e recebimento de dados via stream.

Os diversos contextos criados no sistema, conforme descritos em 5.5.2, utilizam essa camada de acesso externo para carregar os dados necessários. Na inicialização de cada contexto, chamadas de função para esta camada são realizadas, se necessário. Essas chamadas são responsáveis por fazer requisições ao backend e por receber as informações retornadas.

No exemplo abaixo é feito o uso do hook useEffect para chamar uma função que acessa a camada de acesso externo para carregar dados referentes aos sensores, sendo eles os dados em tempo real, e dados dos gráficos, assim que o contexto é inicializado.

```
const fetchSensorData = useCallback(async()=>{
    await Promise.all([
        getRealTimeDataData(reciveRealTimeData),
        fetchGraphData(),
    ])
},[])

useEffect(()=>{
    fetchSensorData()
},[])
```

Requisições HTTP

A biblioteca Axios foi empregada para facilitar a realização das requisições ao backend. Esta biblioteca proporciona uma interface simples e eficiente para criação de requisições

HTTP, e foi integrada nas funções da camada de acesso externo. Essa biblioteca possibilita uma configuração inicial para ser utilizada em todas as requisições realizadas.

Na configuração utilizada, é lido das variáveis de ambiente do sistema, o endereço do backend e a url para adicionar as configurações a rota base, para onde todas as requisição devem ir. Outra configuração aplicada é a adição do interceptor no momento que a requisição é realizada, adicionando no header a configuração necessária para a autenticação, fazendo a leitura do access token do local storage, e adicionando no formato correto para leitura do backend em requisições que precisam de autenticação.

```
const baseUrl = process.env.NEXT_PUBLIC_API_URL
const apiRoute = process.env.NEXT_PUBLIC_API_ROUTE

const baseApi = axios.create({
  baseURL: `http://${baseUrl}${apiRoute}`
});

baseApi.interceptors.request.use(function (config) {
    let token = localStorage.getItem("access_token")
    if (token) {
      config.headers['Authorization'] = `Bearer ${token}`;
    }
    return config;
  }, function (error) {
    return Promise.reject(error);
});
```

As funções dessa camada utiliza essa configuração base para realizar a busca dos dados no backend. Na função abaixo é exemplificado um dessas funções da mama de acesso externo para buscar os dados de um determinado gráfico. A função utiliza a configuração base do `axios` junto com uma url especifica para acesso ao endpoint desejado para buscar

as informações. Se o retorno estiver com o `status code` igual a 200 significa que a requsição foi bem sucedida, então os dados são retornados para o contexto que fez a chamada dessa função. O contexto recebe os dados e disponibiliza para toda a aplicação, como explicado em 5.5.2.

```
async function getGraphData(
  machine:string,
  type:SensorType,
  startDate:Dayjs,
  endDate:Dayjs):Promise<Array<MachineGraphAggregateData>>{
  try{
    let url = "/iot/graph_info"+get_graph_query(
      machine,
      type,
      startDate,
      endDate)
    let response = await baseApi.get(url)
    if(response.status === 200){
      return response.data
    }
    throw("Erro to access graph - "+response.status)
  }catch(ex){
    throw("Erro to access graph - "+ex)
  }
}
```

Conexão WebSocket

Além das requisições HTTP, a camada de acesso externo também gerencia a conexão WebSocket. Especificamente, o contexto de notificações faz uso dessa conexão para receber e

gerir notificações em tempo real.

A gestão da conexão WebSocket é realizada por meio da classe *CustomSocketConnection*. Esta classe é projetada como um singleton, garantindo que uma única instância seja criada e reutilizada em toda a aplicação. Ela é responsável por inicializar e manter o objeto `Socket`, que é parte da biblioteca *socket.io-client*.

```
class CustomSocketConnection{
  private static _instance:CustomSocketConnection|null = null
  private _socketio: Socket|null = null

  private constructor() {
      if (CustomSocketConnection._instance === null) {
          CustomSocketConnection._instance = this;
      }
  }

  public static getInstance(): CustomSocketConnection {
      if (!CustomSocketConnection._instance) {
          CustomSocketConnection._instance = new CustomSocketConnection();
      }
      return CustomSocketConnection._instance;
  }

  get socketio(){
    if(this._socketio===null){
      let token = localStorage.getItem("access_token")
      let url = process.env.NEXT_PUBLIC_API_URL??"localhost"
      this._socketio = io(url, { autoConnect: false, auth: { "Authorization": token } })
    }
```

```
    return this._socketio
  }
}
```

O método `getInstance()` assegura que apenas uma instância da classe seja criada. Essa instância é armazenada como um atributo estático e é retornada sempre que solicitada.

Uma instância da classe *CustomSocketConnection* é utilizada no contexto de notificações. Através dessa instância, o contexto consegue receber mensagens do servidor, manipulá-las e, em seguida, disponibilizá-las para toda a aplicação.

A classe também inclui um mecanismo de autenticação. O token de acesso é recuperado do armazenamento do local storage e é utilizado como parte do cabeçalho de autenticação durante o processo de conexão.

Dentro do contexto de notificações, o atributo Socket da classe é utilizada na inicialização do contexto para realizar a conexão. A instancia da classe é armazenada em uma constante, e em seguinda três funções são cadastradas em eventos de conexão, recebimento de notificação e disconexão. A função que trata o evento de receber uma nova notificação, envia o dado para uma função auxiliar que tem o objetivo de analizar o dado recebido e atualizar a lista de notificações que aparecem na tela.

```
useEffect(() => {
  const socket = CustomSocketConnection.getInstance()
  function onConnect () {
      console.log(‘Connected with id: ${socket.socketio.id}‘);
      console.log(‘Connection Status: ${socket.socketio.connected}‘);
  }

  function newNotification(data:any){
      console.log("Socket Io Notification", data);
      checkNewNotification(data);
```

```
  }

  function disconnect(data:any) {
      console.log("Socket Io disconnect", data);
      console.log("Connection Status");
  }

  socket.socketio.on('connect', onConnect);
  socket.socketio.on('Notification', newNotification);
  socket.socketio.on('disconnect', disconnect);

  return () => {
      socket.socketio.off('connect', onConnect);
      socket.socketio.off('Notification', newNotification);
      socket.socketio.off('disconnect', disconnect);
  };
}, [notifications]);
```

Dessa forma a camada de acesso externo disponibiliza um meio de conexão Web Socket para ser usado para receber novas notificações enquanto o usuário está conectado ao sistema.

Recebendo dados via Stream

Em relação a camada de acesso externo, o último tipo de conexão é o recebimento de dados via stream. Este mecanismo permite a atualização constante dos dados recebidos, garantindo assim um fluxo contínuo de informações atualizadas para a aplicação. Especificamente, essa conexão é usada para disponibilizar os dados dos sensores que são recebidos no backend, como explicado em X.

O recebimento de dados via stream é implementado usando a função *fetch*, nativa do

JavaScript. Essa função é responsável por realizar a requisição ao endpoint correspondente e obter o fluxo de dados em tempo real.

A função `readStream` é utilizada para interpretar os dados do stream. Essa função recebe o leitor do corpo da resposta, e retorna os dados convertidos para o formato `JSON`.

```
async function readStream(
  reader:ReadableStreamDefaultReader<Uint8Array> | undefined) {
  let result = await reader?.read()
  if (!result?.done) {
      let value = result?.value
      if(value){
          const jsonData = parseBytesToJson(value)
          return jsonData
      }
  }
  return false

}
```

A função `getRealTimeDataData` é a responsável por iniciar o processo de streaming de dados. É nessa função que o contexto de dados dos sensores IoT faz a chamada e, consequentemente, recebe e envia os dados para a função recebida como parametro.

Como a função não faz o uso da estrtura do axios explicado em 5.5.3, é necessário realizar a configuração de autenticação da mesma que foi explicado anteriormente na estrutura.

Após realizar a requisição, o leitor dos dados é lido do corpo da resposta da requisição, e passado para a função `readStream`, explicada anteriormente, para ser interpretada e convertida para `JSON`.

```
async function getRealTimeDataData(setData: UpdateDataFromStream) {
```

```
  try{
    let token = localStorage.getItem("access_token")
    let url = baseApi.getUri()
    let response = await fetch(`${url}/iot/realtime`,{headers:{
        Authorization: `Bearer ${token}`,
        'Content-Type': 'application/json',
    },});

    const reader = response?.body?.getReader();
    let result:Array<MachineRealTimeData>|boolean = await readStream(reader)
    if(typeof(result) === "boolean"){
        throw "Error to read stream data"
    }
    do{
      if (Array.isArray(result)) {
          const typedResult = result as MachineRealTimeData[];
          setData(typedResult);
      }
      result = await readStream(reader)
    }while(result!==false)
  }catch(ex){
    console.error(ex)
    throw ex
  }
}
```

A função `getRealTimeDataData` fica sendo executada enquanto a conexão está aberta e não ocorre nenhum erro. O método `setData` é então chamado para enviar o resultado da leitura para o contexto que realizou a chamada da função.

Dentro do contexto de dados dos sensores IOT, a função `setData` passada como parâmetro, apenas atualiza o estado global para atualizar a informação em todos os componentes que fazem uso dela.

5.5.4 Construção dos componentes

Na implementação do sistema, foi utilizada a lógica de construção de componentes para compor a telas. A modularidade e reutilização de código são fatores críticos que motivam essa escolha.

Os componentes acessam dados diretamente dos contextos, conforme discuto em X. Este método facilita a passagem de dados e permite que os componentes sejam mais específicos em sua função, além de evitar a passagem de muitas propriedades na árvore de componentes.

A base para os componentes menores foi extraída da biblioteca Material UI 5 (MUI5). Isso inclui elementos como botões, contêineres, caixas de texto e outros. Um exemplo prático é o menu lateral nas páginas do dashboard, onde o componente Drawer do MUI5 foi empregado.

Um componete que merece destaque é o `Grid` do Material UI 5. A utilização do Grid permitiu uma organização espacial dos elementos da interface do usuário de maneira simples. Esse sistema de grid oferece uma abordagem flexível para alocar espaço, alinhar conteúdo e lidar com variações de tela, o que é especialmente útil em aplicações web com muitas informações e diferentes componentes em tela. No menu lateral por exemplo, foi utilizado o componente `Grid` para organizar as informações do menu e definir o posicionamento de acordo com o estado de aberto ou fechado.

```
<Grid
  container
  direction="column"
  justifyContent="space-between"
  alignItems={open?"center":"start"}
```

```
  height={"97vh"}
>
  <Grid
    container
    item
    direction="column"
    justifyContent="space-between"
    alignItems={open?"center":"start"}
  >
    // Conteúdo
  </Grid>
</Grid>
```

O uso do Grid, portanto, contribuiu para a coesão do layout, fornecendo uma estrutura sólida sobre a qual outros componentes poderiam ser organizados de forma simples, e fácil de entender, cumprindo requisitos de facilitar a manutenção.

Por outro lado, para a construção dos gráficos, a biblioteca Recharts foi utilizada. O gráfico mostrado no dashboard com a informações geradas pelo modulo de processamento, detalhado REF X, é composto por gráficos Scatter, Area, Bar e Line, permitindo assim uma análise multifacetada das informações geradas.

Para componentes que necessitam de manipulação de datas, a biblioteca Days Js foi integrada. Um caso de uso é o componente de filtro de data para exibição de gráficos, que também emprega o DatePicker do MUI5 para uma interface de usuário mais intuitiva. A biblioteca Days Js facilita a manipulação da entrada e saída de dados de data.

```
  <DatePicker
  value={dateFilter.startDate}
  onChange={(value)=>onChangeDate(value, "startDate")}
  label="Data inicial"
  />
```

```
const onChangeDate = (newDate:Dayjs|null,
  dateField:"startDate"|"endDate")=>{
  if(newDate!==null){
    setIsDataUpdated(false)
    if(dateField==="endDate"){
      setDateFilter(oldValue=>({...oldValue,"endDate": newDate}))
    }else{
      setDateFilter(oldValue=>({...oldValue,"startDate": newDate}))
    }
  }
}
```

Capítulo 6

Características do Sistema do ponto de vista funcional

Após o detalhamentos dos requisitos, da arquitetura escolhida para o sistema, das tecnologias utilizadas no projeto e do detalhamento da implementação de cada componente do sistema, esse capítulo é voltado para a explanação das funcionalidades que compõem a aplicação. O projeto abrange diversas componentes, incluindo uma camada de frontend, uma camada de backend, um módulo de recebimento de dados, um módulo de processamento de dados e o banco de dados, portanto, este capítulo tem como objetivo detalhar o funcionamento de cada funcionalidade do sistema, fornecendo uma visão abrangente de como cada componente interage e contribui para a operação do sistema como um todo.

Cada seção deste capítulo se dedicará a uma funcionalidade específica, examinando seu papel e operação em profundidade, bem como a interação entre diferentes componentes do sistema para sua realização. Dessa forma, será possível ter uma compreensão completa de como cada parte do sistema contribui para a funcionalidade geral e os objetivos do projeto.

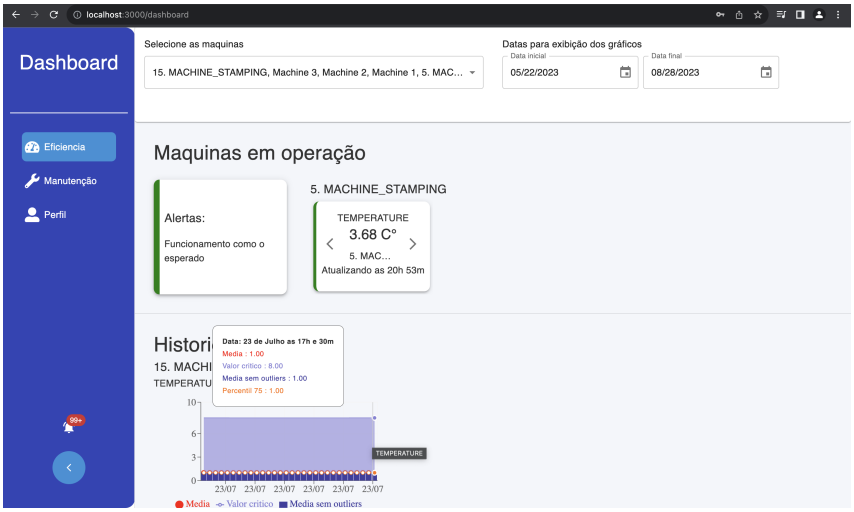

Figura 6.1: Dashboard with real time and graph data.

6.1 Monitoramento em tempo real

No dashboard de interface do usuário, figura 6.1, são apresentados cartões individuais correspondentes a cada máquina monitorada. Em cada cartão, informações sensoriais específicas são exibidas. Um mecanismo de interface intuitivo, ativado pelo clique em uma seta direcional, permite ao usuário expandir as informações para visualizar dados de sensores adicionais de uma máquina específica. O cartão pode ser visto na figura 6.2.

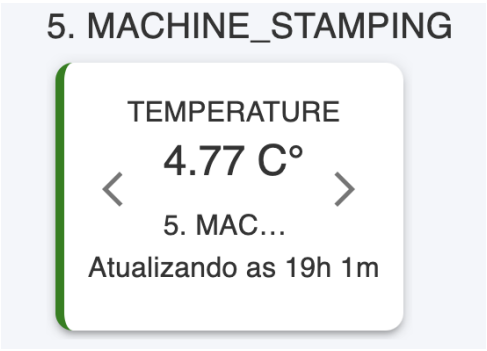

Figura 6.2: Card with machine data.

A obtenção desses dados em tempo real é efetuada através de um fluxo contínuo de dados, conhecido como *streaming*. O frontend da aplicação faz uma requisição ao endpoint `iot/realtime`, o qual retorna esse fluxo de dados em tempo real. O acesso a

essa informação ocorre como explicado em X.

Na camada de backend, o fluxo de dados é constantemente alimentado pela classe `SensorValue`, que por sua vez, é atualizada pelo módulo de recebimento de dados. Ao receber novas leituras dos sensores, o módulo executa validações apropriadas antes de atualizar os valores na classe `SensorValue`, como explicado em X. Uma vez atualizada, a API acessa esses novos valores e os insere no fluxo de dados transmitido ao usuário conectado.

6.2 Alertas e notificações

Esta funcionalidade tem como principal objetivo monitorar o desempenho das máquinas em tempo real e emitir alertas e notificações aos usuários caso sejam detectadas condições de operação inadequadas. Isso permite que ações corretivas sejam tomadas de forma imediata.

Sempre que uma nova leitura de sensor é recebida pelo módulo de recebimento de dados, uma validação é realizada para verificar se a máquina está operando dentro dos parâmetros aceitáveis. Esses parâmetros são obtidos através dos metadados do sistema, explicado em X.

Se um valor fora do intervalo aceitável é identificado durante a validação, o módulo de recebimento de dados insere uma marcação especial nessa leitura dentro da classe `SensorValue`. Essa marcação é posteriormente transmitida ao frontend durante o processo de transmissão de dados via *stream*, como explicado em X.

Ao receber uma leitura marcada, o frontend atualiza o cartão de informação correspondente para refletir o estado anômalo. Especificamente, dentro do dashboard mostrado na figura 6.1, a cor do cartão é alterada para vermelho ou amarelo, tanto nos cards individuais, figura 6.2, quanto no card geral que é usado para mostrar o status geral das maquinas, figura 6.3, portanto, servindo como um alerta visual imediato para o usuário.

O módulo de recebimento de dados, REF, mantém o controle do estado operacional de cada máquina que está enviando informações. Quando uma máquina sai de um estado

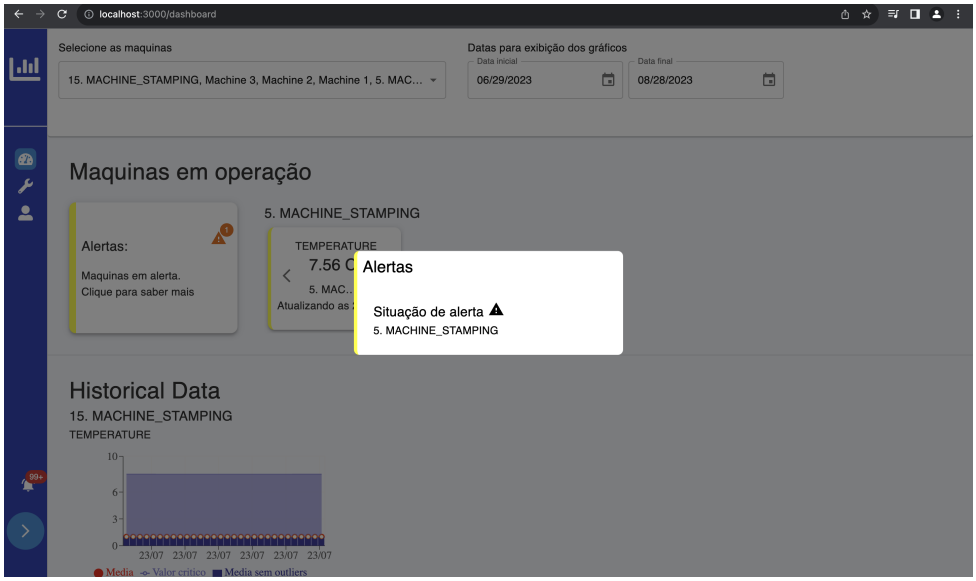

Figura 6.3: General Notifications.

de alerta, a ocorrência é registrada no banco de dados, com data de início e término do estado de alerta, junto com as informações referentes ao sensor e à máquina envolvida, como explicado em X. Posteriormente, uma mensagem é enviada aos usuários conectados utilizando a interface de *WebSocket*, informando-os da mudança de estado e possibilitando uma resposta mais ágil.

As informações sobre a finalização de alertas podem ser acessadas pelos usuários através de notificações na interface do sistema, como pode ser visto na figura 6.4. Ao clicar no ícone de notificação no menu lateral do *dashboard*, o usuário pode visualizar essas notificações.

Portanto, ao iniciar uma sessão no sistema, as notificações anteriores são carregadas para o usuário. Além disso, qualquer nova notificação gerada durante a sessão do usuário é transmitida em tempo real por meio de uma conexão *WebSocket* estabelecida entre o frontend e o backend, como explicado em X.

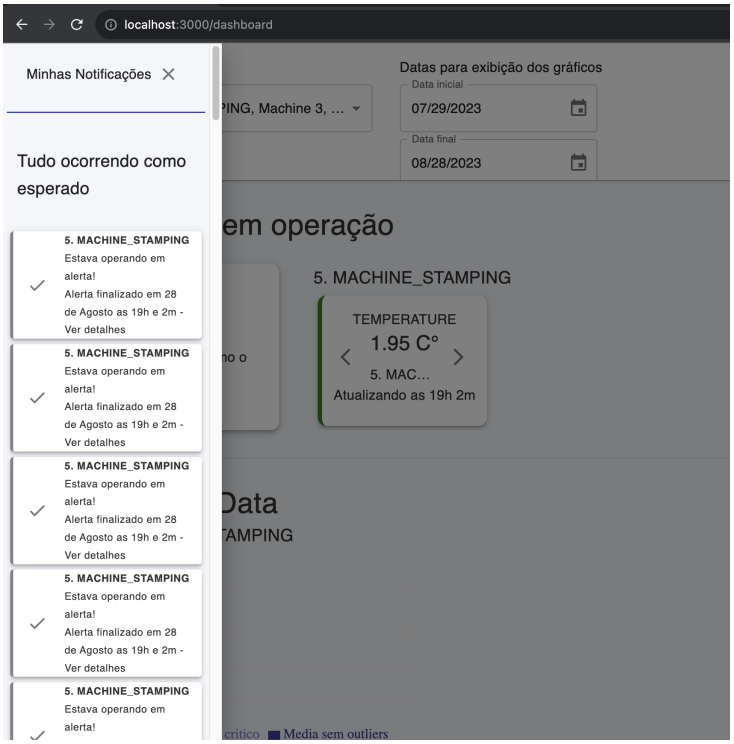

Figura 6.4: Notification drawer.

6.3 Analise estatística de dados históricos

O módulo de processamento de dados é responsável pela análise estatística dos dados históricos. Seguindo o parâmetro padrão do sistema, todos os dias à meia-noite, os dados brutos não processados são lidos e analisados. O resultado da análise é então armazenado no banco de dados para consultas futuras, como explicado em X.

A possibilidade de acessar esses dados analíticos é providenciada pela API, especificamente pelo módulo "IOT Analytics". A requisição para obter essas informações deve ser autenticada, conforme descrito na seção "Módulos da API".

A representação visual dessas análises é efetuada por meio de gráficos que agregam quatro métricas principais resultantes do processamento. Dessa forma é exibido no gráfico:

- *Valor Ideal*: Representado em um gráfico de área, serve como um parâmetro de funcionamento ideal para fornecer uma perspectiva relativa aos outros dados.
- *Percentil 75*: Exibido em um gráfico de linha, esta métrica oferece uma visão sobre

a distribuição dos valores durante o período de agregação e sua evolução ao longo do tempo.

- *Média da Agregação*: Representada em um gráfico de dispersão, esta métrica fornece o valor médio dos dados agregados.

- *Média com Remoção de Outliers*: Ilustrada em um gráfico de barras, esta métrica é calculada após a remoção dos valores *outliers*, conforme determinado pelo método de construção do boxplot.

Portanto, a imagem 6.5 mostra como fica a visualização dessas informações dentro do dashboard.

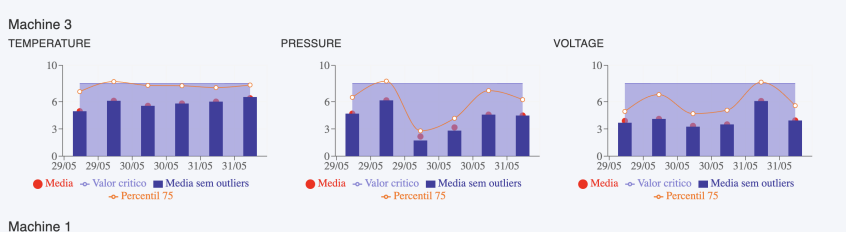

Figura 6.5: Graph example.

6.4 Exibição dos dados referente a paragem das máquinas

Esta funcionalidade é dedicada à exibição de dados relacionados às paragens das máquinas, armazenados previamente no banco de dados. Como esclarecido na seção XX, a funcionalidade tem o objetivo de demonstrar como seriam apresentados esses dados, caso fossem recebidos pelo sistema de forma similar aos dados dos sensores. A fonte dos dados para a elaboração desses gráficos provém de três planilhas recebidas no início do desenvolvimento do projeto, em que as informações foram salvas no banco de dados. Esses dados são acessados pelo frontend por meio do módulo `downtime_analytics` da API.

Os gráficos gerados para representar esses dados são apresentados na forma de gráficos de coluna. Cada gráfico exibe um título correspondente à planilha da qual os dados foram extraídos. A legenda do gráfico indica a porcentagem que cada parada específica representa em relação ao total de paragens.

Os gráficos de coluna fornecem uma maneira eficaz de comparar as diferentes paragens das máquinas, como pode ser visto na figura 6.6. Esta visualização oferece um meio para analisar a eficiência operacional, identificar possíveis áreas para melhoria e acompanhar o resultado das ações tomadas em relação a paragem das máquinas e a manutenção preditiva.

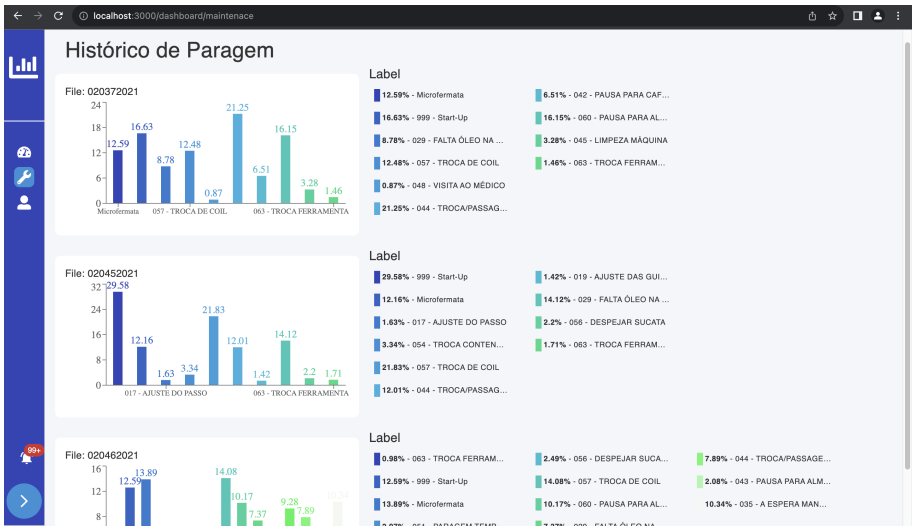

Figura 6.6: Downtime graph.

6.5 Perfil de usuário

A funcionalidade de perfil de usuário foi desenvolvida com o objetivo de fornecer aos usuários acesso aos seus dados pessoais armazenados no sistema. Além da visualização dessas informações, esta tela permite também a realização de modificações, incluindo a alteração de senha, e-mail e outros dados pessoais como nome, sobrenome e descrição.

Para ler e modificar os dados apresentados, o módulo *User* da API é utilizado REF. Este módulo disponibiliza endpoints que permitem o acesso e a modificação dos dados

armazenados, garantindo que as informações sejam atualizadas conforme as interações do usuário, como pode ser visto na figura 6.7.

Um recurso adicional fornecido por esta tela é a opção de logout. Ao selecionar esta opção, todas as informações armazenadas do lado do cliente são limpas, efetuando assim a saída segura do usuário do sistema.

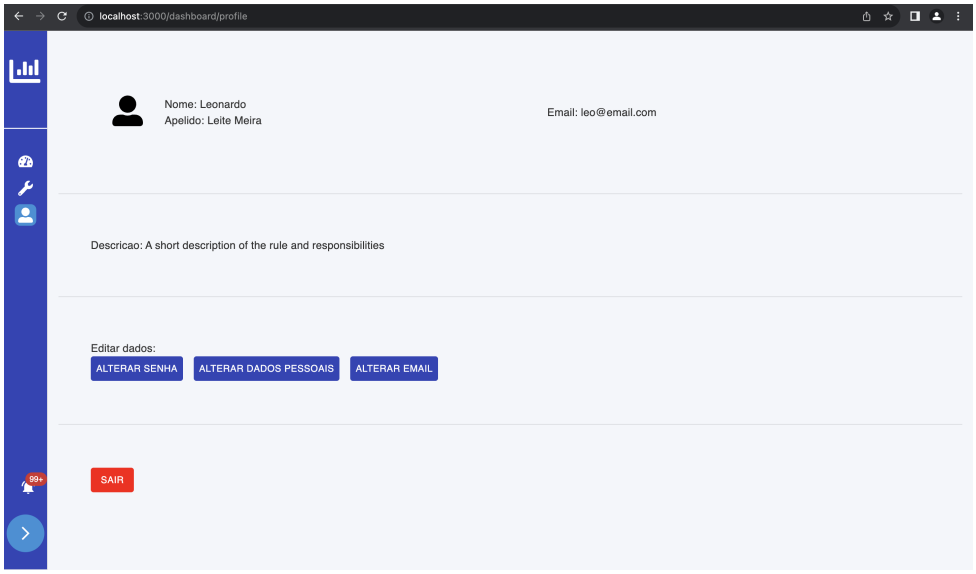

Figura 6.7: Profile page.

Capítulo 7

Resultados e Avaliação

This chapter presents and describes the tests that were developed to check if the project fulfills the objectives and solves the problem described in Analysis/Methodology.

To better understand, the results of each test should be preceded by a description of the test and the expected results.

The work results are commented, including:

- What can be learned from the results?
- What could be done differently?
- What was beyond initial objectives?
- What are the objectives there were not met and why?

Apresentação dos resultados obtidos - Discutir as principais realizações do projeto, incluindo a capacidade de monitorar em tempo real as máquinas da planta e a análise histórica do funcionamento das máquinas.

- Sistema que ajuda no monitoramento das maquinas da planta - Exibir para os funcionários como as maquinas estão funcionando - Gerar insights a partir dos dados gerados

Capítulo 8

Conclusão e Trabalhos Futuros

The conclusions should synthesize and provide a single view to the work developed. It can be done a brief reference to similar work of others and to the knowledge that emerged from it, as well as future work suggestions. The consistency of the document implies that the conclusions should be coherent with the main ideas in the introduction.

8.0.1 Resumo

8.0.2 Limitações do sistema

(Adicionar novas camadas de dados e paginas no dashboard pode ser demorado)?

8.0.3 Sugestões para trabalhos futuros

- melhorias de código - Monitoramento do funcionamento do sistema e logs - Log para recebimento dos dados e quando para de receber - Log para a analise estatistica - Log para erros no recebimento dos dados - Deixar os parâmetros para alertas do sistema dinâmicos, podendo ser alterados pelos usuários adm - Generalização para outros contextos - destacando a tendência crescente de coleta e análise de dados na indústria - Melhoria da performance para grande numero de maquinas e graficos exibidos na tela ao mesmo tempo. - Upgrade para next 13 e 14. Pois com isso podemos ter o beneficio dos server

componentes, o que melhoraia a performace da aplicação - Os graficos podem ser carregados como server componentes com cache equivalente ao intervalo que o modulo de processamento roda, otimizando assim o frontend da aplicação

Apêndice A

Proposta Original do Projeto

Curso de Licenciatura em Engenharia Informática

Projeto 3º Ano - Ano letivo de 2016/2017

<Título do projeto>

Orientador: <Nome do orientador>

Coorientador: <Nome do coorientador>

1 Objetivo

<Objetivo do projeto>

2 Detalhes

<Detalhes que julguem ser necessários>

3 Metodologia de trabalho

<Eventual metodologia de trabalho>

Dimensão da equipa:

Recursos necessários:

Apêndice B

Outro(s) Apêndice(s)

Listagens de código fonte, texto/imagens produzidos por testes complementares, etc.