Erick Amorim Fernandes

Programação Concorrente Aplicada a um Simulador de Alimentação e Escoamento de um Alto-Forno

Viçosa, MG

2023

Erick Amorim Fernandes

Programação Concorrente Aplicada a um Simulador de Alimentação e Escoamento de um Alto-Forno

Monografia apresentada ao Departamento de Engenharia Elétrica do Centro de Ciências Exatas e Tecnológicas da Universidade Federal de Viçosa, para a obtenção dos créditos da disciplina ELT 402 – Projeto de Engenharia II – e cumprimento do requisito parcial para obtenção do grau de Bacharel em Engenharia Elétrica.

Orientador: André Gomes Tôrres

Viçosa, MG

2023

ERICK AMORIM FERNANDES

PROGRAMAÇÃO CONCORRENTE APLICADA A UM SIMULADOR DE ALIMENTAÇÃO E ESCOAMENTO DE UM ALTO-FORNO

Monografia apresentada ao Departamento de Engenharia Elétrica do Centro de Ciências Exatas e Tecnológicas da Universidade Federal de Viçosa, para a obtenção dos créditos da disciplina ELT 402 – Projeto de Engenharia II e cumprimento do requisito parcial para obtenção do grau de Bacharel em Engenharia Elétrica.

Aprovada em 07 de julho de 2023.

COMISSÃO EXAMINADORA

Prof. Dr. André Gomes Torres - Orientador
Universidade Federal de Viçosa

Prof. Dr. Victor Pellanda Dardengo - Membro
Universidade Federal de Viçosa

Me. João Vítor Rodrigues de Vasconcelos - Membro
Universidade Federal de Viçosa

Aos meus amados pais, Maria Aparecida e Maurélio,
e ao meu querido irmão, Yuri.

Agradecimentos

Gostaria de expressar minha profunda gratidão aos meus queridos pais, Maria Aparecida e Maurélio, ao meu amado irmão, Yuri, e à minha tia, Nilceia. Ao longo de toda essa jornada, o apoio incondicional, incentivo constante, carinho e paciência que vocês me proporcionaram foram fundamentais. Nada disso seria possível sem vocês, e é com imensa emoção que deixo registrado todo o amor e respeito que tenho por cada um de vocês.

Além disso, não posso deixar de agradecer ao meu orientador, André Gomes Tôrres. Sua aceitação desta proposta de projeto e sua disponibilidade em esclarecer minhas dúvidas, assim como sua habilidade em ouvir e direcionar minhas ideias, foram essenciais para o sucesso deste trabalho. Sou imensamente grato por ter tido a oportunidade de contar com sua orientação durante todo o processo.

"Remember to look up at the stars and not down at your feet. Try to make sense of what you see and wonder about what makes the universe exist. Be curious. And however difficult life may seem, there is always something you can do and succeed at. It matters that you don't just give up." (Stephen Hawking)

Resumo

Este trabalho tem como objetivo desenvolver um projeto de automação para o controle de alimentação e escoamento de um Alto-forno, utilizando programação concorrente. Além disso, será implementada uma interface web para aquisição de dados textuais, visuais e numéricos. O ambiente de desenvolvimento Microsoft Visual Studio, juntamente com a linguagem de programação *C#*, será utilizado para a criação do software de controle, animação e geração de dados. Os dados serão armazenados em um banco de dados estruturado, PostgreSQL, com acesso através da linguagem Python lançando mão do framework Django. O Django será responsável por criar as rotas de CRUD do serviço web, a rota de comunicação em tempo real utilizando WebSockets e o Redis via WSL, além da página de análise de dados em formato gráfico utilizando HTML e JavaScript. A validação do software será feita por meio da verificação dos dados obtidos durante o processo de automação.

Palavras-chaves: Automação; Alto-forno; Programação concorrente; Alimentação; Escoamento; Tempo real; Python; Django; C#; C Sharp; WebSockets; Threads; PostgreSQL; Redis; JavaScript; Semaphore; Web.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

HTML	HyperText Markup Language
CRUD	Create, Read, Update, Delete
AF	Alto-forno
SQL	Structured Query Language
NoSQL	Not only SQL
C#	CSharp
WSL	Windows Subsystem for Linux
JS	JavaScript
POO	Programação Orientada a Objetos
WPF	Windows Presentation Foundation
DRF	Django Rest Framework
DOM	Document Object Model
CPU	Central Processing Unit
AWS	Amazon Web Services
PSQL	PostgreSQL
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
DRY	Don't Repeat Yourself
ORM	Object-Relational Mapping
API	Application Programming Interface

Sumário

1 Introdução

Nos últimos anos, o avanço tecnológico tem desempenhado um papel fundamental na otimização dos processos industriais, resultando em maior eficiência, produtividade e redução de custos. No setor siderúrgico, especificamente no contexto dos altos-fornos, essa transformação tem sido impulsionada pela implementação de processos de automação em tempo real e pela utilização de sistemas de transporte de cargas eficientes para o abastecimento dessas estruturas.

O alto-forno é uma peça-chave na indústria siderúrgica, responsável pela produção de ferro gusa e outros materiais metálicos. Trata-se de um equipamento de grandes proporções, que opera em altas temperaturas e requer um suprimento contínuo de matérias-primas, como minério de ferro, carvão e calcário (RIZZO, 2009). A eficiência do alto-forno está diretamente relacionada à precisão dos processos de abastecimento e controle de temperatura, pressão e composição química do interior do forno (INFOMET, 1998).

Nesse contexto, a automação em tempo real tem desempenhado um papel crucial, permitindo o controle e monitoramento dos processos industriais de forma precisa e instantânea. A integração de sensores, sistemas de controle e algoritmos avançados possibilita o ajuste automático dos parâmetros operacionais do alto-forno, garantindo um desempenho ótimo e evitando variações indesejadas (OLIVEIRA, 2018).

Além disso, o transporte eficiente das cargas utilizadas no abastecimento do alto-forno também desempenha um papel importante na produtividade e no funcionamento contínuo desse equipamento. Sistemas automatizados de transporte, como correias transportadoras e vagões de carga, permitem o transporte rápido e seguro das matérias-primas, evitando atrasos e interrupções no processo de produção.

No entanto, a simples automatização desses processos não é suficiente para alcançar todo o potencial de melhoria na indústria siderúrgica. A coleta e análise de dados em tempo real, assim como o armazenamento dessas informações, são fundamentais para o monitoramento e otimização contínua do alto-forno. A possibilidade de salvar e acessar esses dados de forma remota, por meio de plataformas web, oferece benefícios significativos, permitindo o acompanhamento em tempo real do funcionamento do alto-forno, o diagnóstico de falhas, a análise de desempenho e a tomada de decisões assertivas.

Por se tratar de um processo que precisa ser constantemente alimentado, este trabalho irá se desenrolar em cima do transporte de material para a alimentação do Alto-forno assim como o controle e a coleta de dados para relatórios de funcionamento do sistema.

1.1 Objetivo

Desenvolver um sistema de alimentação e escoamento para um Alto-Forno, utilizando carrinhos como meio de transporte. Além disso, criar interfaces gráficas e web para o controle e aquisição de dados.

2 Referencial Teórico

Para a produção de qualquer pesquisa se faz necessário uma boa base teórica das ferramentas e dos métodos utilizados, por meio da compilação crítica de várias publicações e/ou documentos relacionados ao tema (AZEVEDO, 1997), além de uma explicação clara, concisa e objetiva para que o leitor acompanhe todo o processo de desenvolvimento do objeto de estudo. Sendo assim, como o projeto lança mão de diversos *softwares*, bibliotecas e padrões de comunicação os tópicos deste capítulo fornecerão toda a base necessária para a compreensão básica do estudo.

2.1 Alto-forno

Um Alto-forno (AF) consiste em uma estrutura industrial que funciona como um forno de grande escala utilizado na produção de ferro fundido. Normalmente possui um formato de torre e sua altura varia entre 30 e 60 metros, possui um interior revestido com materiais refratários altamente resistentes à temperaturas elevadas e à corrosão causada pelos processos químicos decorrentes da produção do Ferro Gusa.

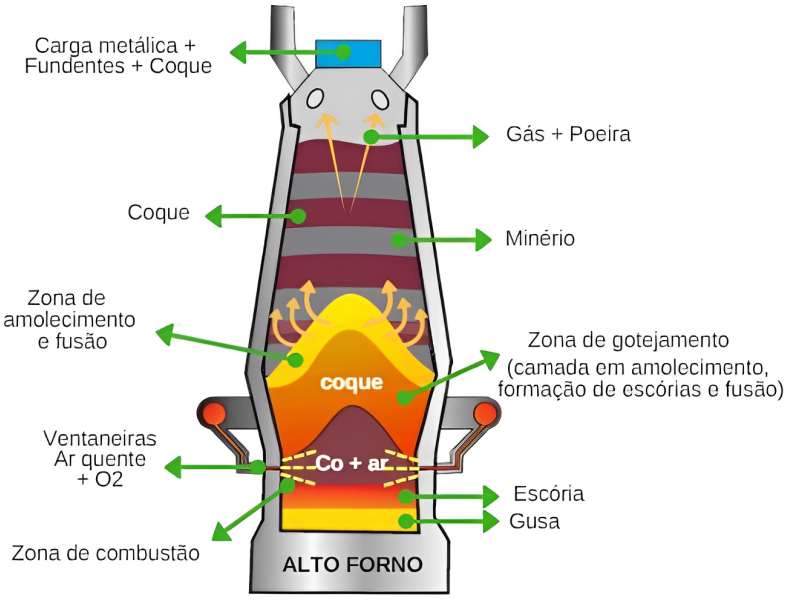


Figura 1 – Esquema simplificado de funcionamento de um Alto-forno(AF).

Fonte: Extraído de (MINERAL, 2018).

Os primeiros AFs conhecidos datam do século XV (RIZZO, 2009) e são sistemas que necessitam de alimentação contínua, comumente realizada por guindantes onde os insumos de minério de ferro são inseridos na parte superior do forno. Durante o processo o AF pode alcançar temperaturas superiores a 1500 graus Celsius e além do ferro líquido produz muitos tipos de resíduos como gases e escórias que podem ser reaproveitadas, respectivamente, como combustível e insumo para a produção de cimento (INFOMET, 1998). A Figura 1 representa um esquema simplificado do funcionamento de um AF.

2.2 Linguagens utilizadas

As linguagens de programação foram criadas para fazer a ponte entre a linguagem humana e a linguagem de máquina, são estruturadas para que programadores consigam comunicar suas intenções ao computador e possuem o papel principal no desenvolvimento de *softwares*, oferecendo aos desenvolvedores as ferramentas fundamentais para traduzir algoritmos em instruções executáveis. Para o desenvolvimento deste projeto foram utilizadas as linguagens *C#*, *Python* e *JavaScript*.

2.2.1 C#

O *C#* (C *Sharp*), Figura 2a, é uma linguagem de programação moderna, orientada a objetos fortemente tipada desenvolvida pela *Microsoft* (MICROSOFT, 2023c). Pode ser utilizada para o desenvolvimento tanto de aplicativos *desktop*, web e móveis além de fazer parte da plataforma .NET.

Dentre suas principais características possui a POO (Pogramação Orientada a Objetos), que permite a criação de classes, objetos e estruturas para modelagem de sistemas. Gerenciamento automático de memória através do coletor de lixo (*Garbage Collector*) que gerencia automaticamente a alocação e liberação de memória. Digitação estática, ou seja, os tipos de dados de todas as varáveis são declarados explicitamente e não podem ser alterados. Acesso ao *Framework .NET* que possui suporte a interfaces gráficas, comunicação em rede, acesso da bancos de dados e etc. Além disso, desde a versão 6.0 do .NET o *C#* passou a ser multiplataforma, podendo ser utilizado tanto no *Windows* quanto no *macOS* e distribuições *Linux*.

O *C#* é amplamente utilizado no desenvolvimento de aplicações contendo interfaces gráficas *desktop* ricas com as plataformas *Windowns Presentation Foundation* (WPF) e *Windows Forms*. Além disso, costuma ser utilizada para a criação de sistemas embarcados, automação, automação de escritório, integração de sistemas e outros, o *C#* continua sendo uma linguagem extremamente popular e de extrema confiabilidade.

Para este projeto o *C#* será o responsável por criar a interface gráfica de comandos e de situação da planta, assim como responsável pela alimentação dos banco de dados

e fornecedor de informações para a comunicação via *WebSocket* que será abordada futuramente.

2.2.2 Python

Python, Figura 2b, é uma linguagem de programação lançada em 1991 sendo sua versão atual e mais famosa o *Python* 3 lançada em dezembro de 2008 (INSTITUTE, 2012). A linguagem é estruturada para seguir as diretrizes e princípio filosóficos conhecidos como "Zen do Python" (PETERS, 2004) que possui a finalidade de tornar o desenvolvimento fácil, claro e legível.

Por ser uma linguagem moderna e de fácil entendimento é considerada uma das mais queridas pelos programadores e com uma das maiores comunidades no mundo. Além de possuir uma gigantesca biblioteca padrão que oferece uma grande gama de módulos e funções prontos para o uso, o *Python* possui uma infinidade de bibliotecas de terceiros altamente respeitadas e utilizadas como o *Numpy*, *Pandas*, *TensorFlow* e *Django*, que inclusive será utilizada neste projeto.

Ademais o *Python* é extremamente flexível podendo ser utilizado tanto para diversas áreas como aprendizado de máquina, processamento de imagens, ciência de dados e desenvolvimento web, tanto quanto para criação de jogos, aplicações mobile, aplicações desktop e automação de tarefas.

Por ser tão bem avaliada por seus usuários o *Python* é utilizado por grandes empresas como *Google*, *Facebook*, *Netflix* e *Dropbox* (REYNOLDS, 2018) em seus produtos e serviços, além de ser a linguagem preferida dentro de *DataScience* (INSIGHTLAB, 2019).

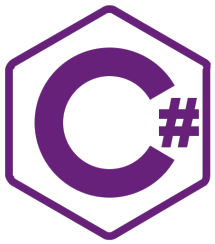
Neste programa o *Python* será o responsável pelas comunicações web utilizando o *framework Django*, pela criação de rotas de comunicação em tempo real via *WebSocket* e pelas rotas que retornam informações contidas no banco de dados utilizando o padrão *REST*.

2.2.3 JavaScript

O *JavaScript*(JS), Figura 2c, é uma linguagem amplamente utilizada para o desenvolvimento web (MDN, 2023b). Ela permite o desenvolvimento de páginas interativas e dinâmicas, sendo excelente em interagir com o cliente, possibilitando animações, validação de documentos e manipulação de elementos em páginas html. Uma de suas características é o DOM (*Document Object Model*) (MDN, 2023a) que habilita a possibilidade de acessar e alterar elementos da página html em tempo real assim como a realização de requisições assíncronas que permite às paginas serem atualizadas apenas em partes especificas. Além disso, o JS oferece ferramentas para armazenagem local de dados no navegador do usuário e para manipular vídeos, áudio e gráficos.

Assim como o *C#* e o *Python* o JS permite o uso de POO (Programação Orientada a Objetos). Pode ser utilizada tanto para o desenvolvimento de sites quanto para o *desktop* e para o *mobile*. É uma linguagem clássica e de peso no mercado e que possui muitos usuários, assim como uma grande comunidade.

Neste projeto o JS será utilizado em um *template* html de um rota configurada no *Django*, o código JS se encontra no template html da rota e será responsável por plotar gráficos com o valores recebidos do banco de dados.



(a) C# (C Sharp)

(b) Python

(c) JavaScript

Figura 2 – Logotipo das linguagens utilizadas.

Fonte: Autor.

2.3 Programação Assíncrona (Multithreading)

A programação *multithread* é utilizada em situações em que existem tarefas independentes e paralelizáveis que podem ser executadas de forma simultânea visando uma melhora no desempenho e, consequentemente, uma melhora na eficiência do programa (MICROSOFT, 2023b). Um dos maiores benefícios da programação *multithreading* está no fato de que mesmo em processos independentes ainda existe comunicação entre as *threads*, que compartilham dados e recursos.

Entretanto, ao compartilhar dados e recursos, podem ocorrer problemas de concorrência, como a modificação de uma variável, o que pode gerar inconsistências no dado modificado, sendo assim necessário mecanismos de sincronização, como *mutex* e *semaphore* no caso do *C#*. Outro ponto importante a se destacar está no fato de que a programação *multithreading* pode apresentar problemas de escalabilidade, bloqueios e condições de corrida, portanto para sua implementação se faz necessário um planejamento minucioso e um bom entendimento dos conceitos de sincronização e gerenciamento de *threads*.

Para um melhor entendimento do tópico vamos abordar os conceitos e o gerenciamento de *threads* nos itens a seguir.

2.3.1 Threads

Um processo é um programa em execução no sistema operacional. Ele serve para separar os diferentes aplicativos que estão sendo executados, garantindo que cada um tenha recursos e memória separados.

Uma *thread*, por sua vez, é a unidade básica de alocação de tempo do processador pelo sistema operacional. Cada *thread* possui uma prioridade de agendamento e mantém um conjunto de estruturas que são usadas pelo sistema para salvar o contexto da mesma quando pausada. O contexto da *thread* inclui todas as informações necessárias para que ele possa continuar sua execução sem interrupções, como os registros de CPU e a pilha de execução (MICROSOFT, 2023b).

Dentro de um processo, é possível ter várias *threads* em execução. Todos as *threads* compartilham o mesmo espaço de endereço virtual do processo, o que significa que elas podem acessar e executar qualquer parte do código do programa, inclusive trechos que estejam sendo executados simultaneamente por outras *threads* dentro do mesmo processo.

2.3.2 Mutex e Semaphore

O objeto *mutex* (abreviação para "mutual exclusion") é um mecanismo de sincronização do *C#* que permite que apenas uma *thread* acesse um recurso compartilhado por vez. Quando uma *thread* adquire um *mutex*, ela bloqueia outras *threads* de acessar o recurso até que ela mesma o libere. O *mutex* garante que apenas uma *thread* execute uma seção crítica de código por vez, evitando condições de corrida (situações em que múltiplas *threads* acessam o mesmo recurso simultaneamente, resultando em resultados indefinidos ou inconsistentes). O uso adequado de *mutexes* ajuda a garantir a consistência e a integridade dos dados compartilhados (MICROSOFT, 2023).

Ao contrário de um *mutex*, que permite o acesso exclusivo a um recurso, um *semaphore* pode controlar o acesso simultâneo a um número específico de recursos. O *semaphore* mantém um contador interno que rastreia o número de recursos disponíveis. As *threads* podem solicitar acesso a um recurso usando o *semaphore*, e o *semaphore* gerencia o acesso, concedendo-o se o recurso estiver disponível ou bloqueando a *thread* até que o recurso esteja disponível. Os *semaphores*, podem ser usados para resolver problemas de sincronização complexos, como a exclusão mútua e a coordenação de várias *threads* (MICROSOFT, 2023a).

As principais características de cada método se encontram na Tabela 1 abaixo.

Tabela 1 – Características do Mutex e do Semaphore. (THAKUR, 2022)

Parâmetro	Mutex	Semaphore
Mecanismo utilizado	Mutex usa um mecanismo de bloqueio.	Semaphore usa um mecanismo de sinalização.
Tipo de dado	Um mutex é um objeto.	Um Semaphore é uma variável inteira.
Modificação	Um mutex só pode ser modificado pelo processo que solicita ou libera um recurso.	As operações de espera (wait) e sinalização (signal) podem modificar um Semaphore.
Número de threads	Múltiplas threads de programa em um mutex podem existir, mas não simultaneamente.	Múltiplas threads podem existir.
Operação	O objeto Mutex está ou bloqueado ou desbloqueado.	As operações Wait() e signal() modificam o valor do semaphore de operação.
Gerenciamento de recursos	Se o mutex estiver bloqueado, o processo deve esperar e ser mantido em uma fila.	Se nenhum recurso estiver disponível, a operação de espera (wait) é executada pelo processo.
Tipos	Não existem subtipos.	Existem dois tipos de semáforos: semáforo de contagem (counting semaphore) e semáforo binário (binary semaphore).
Ocupação de recursos	Se o objeto mutex já estiver bloqueado, o processo que solicita recursos aguarda até que o bloqueio seja liberado e é colocado em fila pelo sistema	Se todos os recursos estiverem sendo usados e o processo que solicita recursos executar a operação wait() no semáforo, ele se bloqueará até que a contagem do semáforo se torne maior que 0, indicando que um recurso está disponível

2.4 Bancos de dados

2.4.1 SQL (Postgres)

A Linguagem de Consulta Estruturada (SQL) é uma linguagem de programação projetada para armazenar e manipular informações em um banco de dados relacional. Nesse tipo de banco de dados, as informações são organizadas em tabelas, com linhas e colunas representando atributos de dados e as relações entre os valores. Por meio de instruções SQL, é possível armazenar, atualizar, remover, pesquisar e recuperar informações do banco de dados. Além disso, a SQL também permite a manutenção e otimização do desempenho do banco de dados (AWS, 2023).

O *PostgreSQL* (PSQL), figura 3a, é um sistema de banco de dados objeto-relacional

de código aberto altamente poderoso. Ele utiliza e expande a linguagem SQL, proporcionando uma ampla gama de recursos para armazenar e lidar de forma segura com cargas de trabalho de dados complexas. O *PostgreSQL* tem suas raízes no projeto *POSTGRES*, iniciado em 1986 na Universidade da Califórnia em Berkeley, e desde então tem sido continuamente desenvolvido por mais de 35 anos na plataforma principal (POSTGRESQL, 2023).

Outra vantagem significativa do PSQL é sua ampla compatibilidade com padrões SQL. Ele adere rigorosamente às especificações do SQL, tornando a migração de aplicativos de outros sistemas de banco de dados para o *PostgreSQL* mais fácil e eficiente.

O PSQL é uma escolha popular entre desenvolvedores e empresas devido à sua natureza de código aberto e à ativa comunidade de desenvolvedores que o suporta e contribui para aprimoramento constante do sistema, fornecendo atualizações regulares, correções de bugs e novos recursos.

2.4.2 NoSQL (Redis)

Conhecidos como "não relacionais", "*BDsNoSQL*"ou "não SQL", os bancos dados *NoSQL* se destacam por conseguirem processar grandes volumes de dados não estruturados e em mudança constante sem o uso de linhas e tabelas como em bancos SQL. Apesar de existir desde os anos 60, os bancos de dados "não relacionais" têm experimentado um aumento significativo em popularidade nos últimos tempos. Esse aumento está relacionado à mudança no panorama dos dados, à necessidade dos desenvolvedores de lidar com grandes volumes e uma ampla variedade de informações geradas na nuvem, dispositivos móveis, redes sociais e *Big Data*. A cada dia que se passa, os bancos *NoSQL* vem passando por evoluções que auxiliam os desenvolvedores na rápida criação de sistemas de bancos de dados, visando o armazenamento ágil de novas informações e a sua pronta disponibilização para fins de pesquisa, consolidação e análise (AZURE, 2023).

O *Redis*, Figura 3b, é um banco de dados *NoSQL* em memória, de código aberto (licença BSD), usado como banco de dados, *cache*, *message broker* e mecanismo de *streaming*. O *Redis* fornece estruturas de dados como *strings*, *hashes*, listas, conjuntos, conjuntos ordenados com consultas de intervalo, *bitmaps*, *hyperloglogs*, índices geoespaciais e *streams*. Possui, também, replicação incorporada, *scripts Lua*, evicção LRU, transações e diferentes níveis de persistência em disco, além de oferecer alta disponibilidade por meio do *Redis Sentinel* e particionamento automático com o *Redis Cluster* (REDIS, 2023).

Além disso é possível executar operações atômicas nesses tipos, como anexar a uma *string*, incrementar o valor em um *hash*, inserir um elemento em uma lista, calcular a interseção, união e diferença de conjuntos ou obter o membro com a classificação mais alta em um conjunto ordenado. Graças ao seu robusto projeto, suporta replicação assíncrona,

com sincronização rápida e sem bloqueio e reconexão automática com ressincronização parcial em caso de divisão de rede além disso é excelente para aplicações *realtime* que se comunicam via *WebSocket*.

(a) PostgreSQL (b) Redis

Figura 3 – Logotipo dos bancos de dados utilizadas.

Fonte: Autor.

2.5 WebSockets e a comunição em tempo real

WebSocket é um protocolo que usa conexões tcp para comunicação bidirecional em tempo real entre um servidor e um cliente, tornando-a uma ferramenta indispensável para aplicações que exigem atualizações em tempo real. Ao contrário do protocolo HTTP tradicional, onde a comunicação é baseada em solicitação e resposta, os *WebSockets* oferecem uma conexão persistente que permite o envio e recebimento contínuo de dados entre o servidor e o cliente (MELNIKOV; FETTE, 2011).

A principal vantagem do *WebSocket* é a sua capacidade de fornecer uma comunicação em tempo real eficiente e de baixa latência. Em vez de depender de solicitações periódicas do cliente para atualizar informações, o servidor pode enviar atualizações imediatamente assim que os dados mudam, permitindo que os clientes recebam e exibam as informações atualizadas em tempo real. Isso é especialmente útil em cenários em que várias pessoas estão interagindo simultaneamente com os mesmos dados, como salas de bate-papo, aplicativos de colaboração em tempo real, jogos online e monitoramento em tempo real (MELNIKOV; FETTE, 2011).

Além disso, o protocolo *WebSocket* foi projetado para serem amplamente compatíveis com as tecnologias existentes na web. Eles são suportados por todos os principais navegadores modernos e têm bibliotecas disponíveis para a maioria das linguagens de programação populares, o que facilita a sua implementação em diferentes tipos de aplicativos.

A implementação de *WebSockets* geralmente envolve duas partes principais: o servidor e o cliente. O servidor *WebSocket* é responsável por lidar com as conexões e enviar dados para os clientes, enquanto o cliente *WebSocket* é responsável por estabelecer uma conexão com o servidor e receber os dados enviados, observe a Figura 4 para um melhor entendimento.

No entanto, é importante considerar que a utilização de *WebSockets* também apresenta alguns desafios. O principal é a escalabilidade, uma vez que a comunicação bidirecional contínua pode gerar uma carga significativa nos servidores. Portanto, é essencial projetar adequadamente a infraestrutura e considerar a distribuição da carga para garantir que o sistema possa lidar com um grande número de conexões simultâneas.

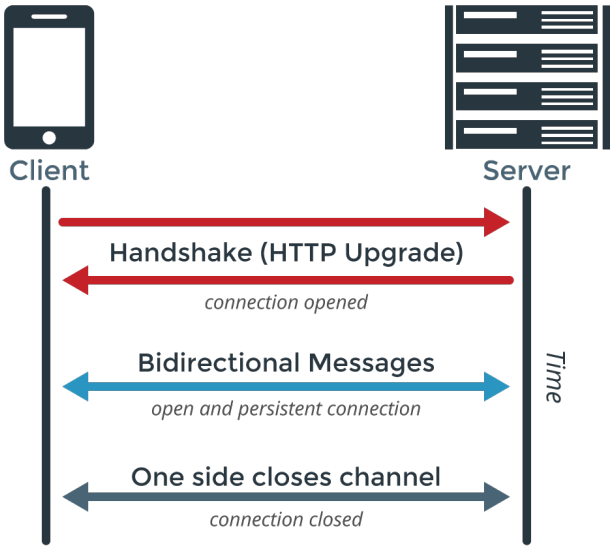


Figura 4 – Esquema simplificado de funcionamento de uma comunicação via WebSocket.

Fonte: Extraído de (MONITOR, 2013).

2.6 Django e Django REST

O *Django* é um *framework* web de alto nível, escrito em *Python*, que permite o desenvolvimento rápido e eficiente de aplicações web. Lançado em 2005, o *Django* foi projetado para facilitar a criação de sites complexos e escaláveis, seguindo o princípio do "não repita a si mesmo"(DRY - *Don't Repeat Yourself*) e promovendo uma abordagem pragmática para o desenvolvimento web (HOLOVATY; KAPLAN-MOSS, 2009).

Com o *Django*, os desenvolvedores podem construir aplicações web robustas e seguras, tirando proveito de uma vasta gama de funcionalidades embutidas. Ele oferece um conjunto completo de ferramentas para ajudar na manipulação de URLs, *templates*,

autenticação de usuários, gerenciamento de sessões, manipulação de formulários, entre outros aspectos cruciais para o desenvolvimento web.

Uma das principais características do *Django* é o seu sistema de ORM (*Object-Relational Mapping*), que permite aos desenvolvedores interagirem com o banco de dados usando objetos *Python*, em vez de escrever consultas SQL manualmente, o que simplifica muito o processo de manipulação dos dados e facilita a portabilidade do código entre diferentes bancos de dados (FORCIER; BISSEX; CHUN, 2008).

Outro destaque do *Django* é a sua comunidade ativa e engajada. A documentação oficial é abrangente e bem estruturada, fornecendo exemplos detalhados e guias passo a passo para facilitar o aprendizado e o uso do *framework*. Além disso, existem uma infinidade de pacotes e bibliotecas de terceiros disponíveis, desenvolvidos pela comunidade, que podem estender ainda mais as capacidades do *Django*.

Uma das extensões mais utilizadas do *Django* é o *Django REST*, projetado especificamente para desenvolvimento de *APIs RESTful*. Ele fornece um conjunto de ferramentas e recursos que simplificam a criação de *APIs* robustas e escaláveis. O *Django REST* possui uma arquitetura flexível e modular, permitindo que os desenvolvedores construam *APIs* personalizadas de acordo com as necessidades específicas do projeto. Ele oferece recursos como serialização de dados, autenticação e autorização, suporte a diferentes formatos de resposta (JSON, XML, etc.), manipulação de solicitações HTTP (GET, POST, PUT, DELETE) e muito mais (FORCIER; BISSEX; CHUN, 2008). Com sua documentação abrangente e uma comunidade ativa, o *Django REST* é uma escolha popular para o desenvolvimento de *APIs* poderosas e eficientes.

2.7 WSL

Apesar de ser uma aplicação multiplataforma, o *Redis*, atualmente, não possui suporte para o *Windows* (MONITOR, 2023). Portanto, para conseguir usa-lo neste projeto foi feito o uso do WSL.

O WSL (*Windows Subsystem for Linux*) é um recurso desenvolvido pela *Microsoft* para permitir que os usuários executem distribuições *Linux* no sistema operacional *Windows*. Ele fornece uma camada de compatibilidade entre os dois sistemas, permitindo que programas e comandos Linux sejam executados no *Windows* sem a necessidade de uma máquina virtual separada.

O WSL permite que você instale e execute uma distribuição *Linux* completa, como *Ubuntu*, *Debian*, *Fedora*, etc., diretamente no *Windows*. Isso permite que os desenvolvedores usem ferramentas e comandos do *Linux* em um ambiente *Windows* sem a necessidade de dual *boot* ou máquinas virtuais (MICROSOFT, 2023).

Ele oferece uma experiência nativa do Linux no *Windows*, permitindo que você execute aplicativos e ferramentas de linha de comando do *Linux* diretamente no *prompt* de comando do *Windows* ou em terminais de terceiros, como o *Windows* Terminal.

3 Metodologia

Neste tópico, será abordado o planejamento e a execução de cada parte do projeto. No entanto, por motivos de organização, o código em si não será mostrado neste documento. Ele estará disponível para consulta em (FERNANDES, 2023a). Nosso foco principal será na explicação da lógica envolvida em cada etapa.

Embora este texto tenha sido elaborado de maneira a não depender do acesso ao código para o seu entendimento, recomenda-se ler este trabalho com o código disponível, pois isso ampliará significativamente a experiência de leitura e compreensão do projeto.

3.1 Regras de funcionamento

Este projeto irá criar um sistema de abastecimento de um Alto-forno seguindo as seguintes regras:

- Carrinhos com os insumos deverão percorrer um percurso com o material, antes de abastecer o alto-forno.
- Carrinhos com as escórias deverão percorrer um percurso diferente com o material.
- Os carrinhos de escória e de insumos se interceptarão em um cruzamento.
- No cruzamento só é permitido um carrinho por vez.
- O programa deve conter interface gráfica que contenha:
  - Animação dos carrinhos.
  - Animação do Alto-forno.
  - Botões de controle.
  - Informações sobre o processo.
- Informações importantes deverão ser salvas em um banco de dados.
- Existir rotas Web para adquirir informações do banco de dados.
- Existir uma rota que envie as mesmas informações contidas na interface para uma página Web em tempo real.

3.2 Desenvolvimento do programa e interface gráfica

Devido a facilidade em construir aplicações com interfaces gráficas oferecido pelo *Forms*, o *C#* foi a linguagem escolhida para esta etapa.

Esta aplicação será a parte do projeto responsável pelo controle e animação dos processos, assim como a alimentação do banco de dados SQL com as informações sobre o sistema e o envio de informações para o *WebSocket* da aplicação *RealTime*.

3.2.1 Geração e movimentação das imagens

Para a criação do *layout* da interface foi criado um modelo *Forms* do *C#* conforme demonstrado na Figura 5.

Observe que foram criados objetos do tipo *"Label"* para os campos responsáveis por informações textuais, indicados como 1. Os objetos do tipo *"Button"*, representados como 2, são responsáveis pelos botões de comandos. Os objetos do tipo *"PictureBox"*, presentes em 3 e 4, são os campos que delimitam onde a criação das imagens será permitida. Por fim, o objeto do tipo *"ListBox"* em 5 é responsável por atualizar as informações de cada carrinho de insumo quando ele finaliza o abastecimento do Alto-forno.

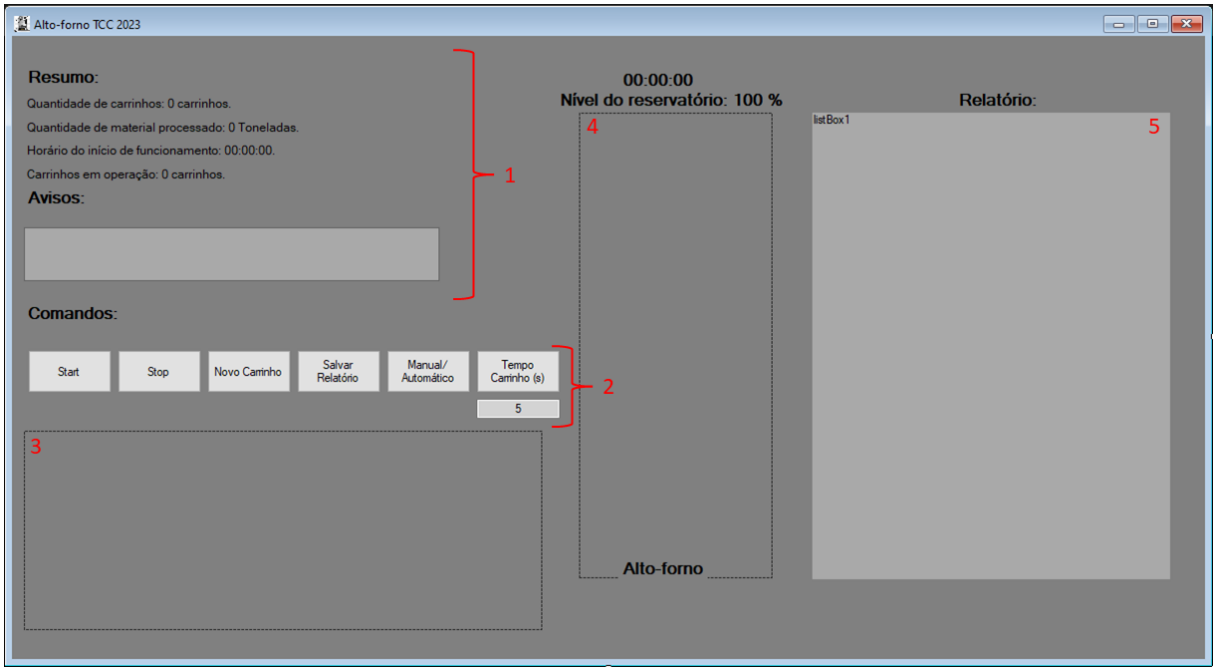


Figura 5 – Interface gráfica desenvolvida via *C#*.

Fonte: Autor.

A parte responsável pelos desenhos segue uma lógica específica. Primeiramente, para preparar a área em que será feito o desenho, utilizamos a função *"new Bitmap"*, que

recebe as dimensões da *PictureBox*. O objeto *Bitmap* é uma imagem em memória que será usado para desenhar o alto-forno. Em seguida, criamos um objeto *Graphics* que recebe o *Bitmap*. O *Graphics* é responsável por desenhar a imagem.

Utilizamos a função Clear com a cor cinza para aplicar o fundo do desenho. Depois disso, definimos alguns objetos do tipo *Pen*, *SolidBrush* e *LinearGradientBrush*. Esses objetos são responsáveis, respectivamente, por criar as formas do desenho, preencher objetos com a cor preta e preencher o nível do reservatório.

Para o preenchimento utilizando o *LinearGradientBrush*, utilizamos as cores vermelha e amarela como referência, a fim de transmitir a sensação de calor. Por fim, utilizamos o método *BackgroundImage* do objeto *PictureBox* para exibir a imagem desenhada. Essa lógica é seguida para todos os desenhos realizados.

Quanto à parte de animação, os objetos são criados e atualizados com base em valores controlados pelo *C#* ao longo do tempo. A cada atualização de tempo, novos valores são calculados e aplicados. Quando esses valores são utilizados nas funções responsáveis pelo desenho, a imagem antiga é substituída pela nova, criando assim a sensação de movimento. Essa representação visual mostra em que estágio o processo se encontra no momento.

3.2.2 Configurações de inicialização

Para garantir o bom funcionamento do projeto, é essencial criar variáveis que serão modificadas para ajustes e o correto funcionamento da aplicação. Alguns dos tipos de variáveis utilizados no projeto incluem:

- *Datetime*: utilizada para recuperar dados como hora de início, hora de término e relógio.
- Variáveis inteiras: utilizadas para modificar as posições dos carrinhos, nível do reservatório, quantidade de carrinhos em movimento, quantidade total de carrinhos lançados, contador de segundos inteiros, entre outros.
- Variável do tipo *Color*: responsável por atribuir cores diferentes a cada carrinho, facilitando a visualização na interface gráfica.
- Variáveis *float*: responsáveis pela movimentação.
- Variáveis *booleanas*: funcionam como *flags* para liberar ou bloquear determinados recursos.
- Variáveis *thread*: responsáveis pelo funcionamento dos carrinhos, onde cada carrinho será uma *thread* independente. Inicialmente foram definidos 999 carrinhos para a entrega de material e 500 para a retirada de escória.

Além disso, algumas regras foram convencionadas para o projeto:

- O alto-forno processará o material a uma velocidade de 1 tonelada por segundo.
- Os carrinhos com insumos transportarão 10 toneladas cada e terão uma velocidade de 10 *pixel* por segundo.
- Os carrinhos de escória terão uma velocidade de 5 *pixels* por segundo e serão ativados a cada 20 toneladas de material processado.
- Para o controle automático, foi definido um valor inicial de 1 carrinho a cada 5 segundos.

Adicionalmente, foram definidas mensagens de aviso para o usuário, levando em consideração o nível do reservatório:

- Se o reservatório estiver acima de 67%: "Nível do reservatório dentro do desejável".
- Se o reservatório estiver entre 67% e 33%: "Nível do reservatório abaixo do esperado, adicione mais material".
- Se o reservatório estiver abaixo de 33%: "Nível do reservatório muito abaixo do esperado, adicione mais material urgentemente".

É importante ressaltar que o programa funciona em sincronia com *timers*. Esses *timers* controlam a velocidade de atualização de cada processo, como a esteira, o alto-forno, a velocidade dos carrinhos, entre outros. Eles são fundamentais tanto para sincronização quanto para simular o papel dos sensores em uma aplicação real.

3.2.3 Botões e suas funções

Foram criados vários botões, conforme representado na Figura 5.2, para promover o controle do sistema. São eles:

- Start: Responsável por acionar os timers do alto-forno e dos carrinhos. Além disso, prepara os carrinhos para serem utilizados, desenhando-os e deixando-os prontos para serem acionados.
- Stop: Interrompe a fila dos carrinhos de insumos, mas permite que os carrinhos de escória continuem até que todo o material do reservatório seja esgotado.
- Novo Carrinho: Envia manualmente um novo carrinho. Caso o sistema esteja no modo automático, essa ação não é permitida e um aviso é exibido.

- Salvar Relatório: Cria um arquivo .txt com todos os valores contidos na *listbox* do relatório.
- Manual/Automático: Alterna entre o envio automático e manual dos carrinhos.
- Tempo do Carrinho (s): Responsável por definir o intervalo de tempo entre o envio de cada carrinho no modo automático. O valor é definido na caixa de digitação abaixo do botão, onde o valor "x" representa um carrinho a cada "x" segundos.

3.2.4 Cruzamento entre escoria e insumos

Conforme estabelecido nos objetivos, no cruzamento é permitida apenas a passagem de um carrinho por vez. Para garantir isso, foi definido que o método *Semaphore* será acionado com apenas uma vaga disponível durante este intervalo específico. Dessa forma, quando um carrinho entrar no espaço, ocupará essa vaga. Quando outro carrinho se aproximar do cruzamento, sua *thread* verificará se há vagas disponíveis. Caso não haja, o processo aguardará até que o espaço seja liberado para que ele possa prosseguir. Os carrinhos que chegarem formarão uma fila de espera. É importante ressaltar que, como apenas uma vaga está sendo disponibilizada, a lógica para o uso do *Mutex* seria exatamente a mesma.

3.2.5 Sobreposição dos carrinhos

Um problema surgiu após a implementação do *Semaphore*: os blocos dos carrinhos se sobrepunham quando chegavam ao cruzamento, pois eles não tinham conhecimento da existência um do outro. Essa sobreposição estava correta dentro da lógica das *threads*, pois todas elas verificavam se havia espaço livre na mesma posição. No entanto, essa sobreposição prejudicava a representação mais realista da simulação.

Para contornar esse problema, foi criada uma rotina que verifica se o valor da *thread* responsável pelo carrinho anterior é pelo menos 6 *pixels* maior do que o valor da *thread* atual. Caso seja, o contador de posição pode ser acionado. Caso contrário, a *thread* aguardará o próximo *tick* do *timer* para verificar novamente e poder avançar caso o carrinho anterior tenha avançado. Isso garante que, como cada bloco de representação de um carrinho tem 5 *pixels* de tamanho, sempre haverá pelo menos 1 *pixel* de separação entre eles.

3.3 Django

O *Django*, por ser um *framework* para desenvolvimento web, será o responsável pela comunicação com o banco de dados, *websockets* e criação das aplicações web. Assim, abordaremos suas principais configurações e funcionalidades neste tópico.

3.3.1 Configuração

Para configurar o *Django*, foi criado um projeto que servirá como base para as aplicações subsequentes. Nesse projeto, foram instaladas algumas bibliotecas que estão listadas no arquivo *requirements.txt*, link disponível em (FERNANDES, 2023b). Essas bibliotecas são responsáveis pelo funcionamento das aplicações desenvolvidas. Em relação à configuração do projeto, no arquivo *settings.py*, no campo "INSTALLED_APPS", foram adicionadas as aplicações instaladas e as que foram desenvolvidas, no caso, as aplicações "chat" e "info".

Além disso, foram criados os campos "STATIC_URL", "MEDIA_URL", "STATIC_ROOT"e "MEDIA_ROOT". O campo "STATIC_URL" define a URL base para os arquivos estáticos do aplicativo, o campo "MEDIA_URL" define a URL base para os arquivos de mídia enviados pelos usuários, o campo "STATIC_ROOT" especifica o diretório de destino onde os arquivos estáticos do projeto serão coletados quando o comando *collectstatic* for executado e o campo "MEDIA_ROOT" especifica o diretório de destino onde os arquivos de mídia enviados pelos usuários serão armazenados. Embora essas configurações não sejam utilizadas neste projeto em particular, são boas práticas organizacionais.

Adicionalmente, foi necessário criar duas configurações específicas para o Channels: "ASGI_APPLICATION" e "CHANNEL_LAYERS". A configuração "ASGI_APPLICATION" é responsável por definir qual serviço "ASGI" será utilizado, enquanto a configuração "CHANNEL_LAYERS" especifica a rota a ser utilizada.

Para finalizar o arquivo *settings.py*, foi adicionada uma configuração para o *Swagger*. Essa configuração é responsável por auxiliar na descrição, consumo e visualização de serviços de uma *API REST*. No projeto, a configuração foi feita para especificar a classe a ser usada como esquema automático padrão para a geração da documentação *Swagger*, utilizando a biblioteca *drf-yasg*.

Por fim, foram adicionadas as referências para as rotas de cada aplicação no arquivo *url.py*. Além disso, foi criado o arquivo *routing.py*, o qual desempenha a função de comunicação com o *WebSocket*.

3.3.2 Banco de dados

Foi elaborado um modelo para armazenar e manter o histórico detalhado do funcionamento do sistema no banco de dados, conforme ilustrado na Figura 6. O modelo consiste nas seguintes tabelas:

- Tabela "info_resumooperacao": Responsável por armazenar os dados iniciais de cada operação do alto-forno. Cada vez que um novo processo é iniciado, um valor é adicionado a essa tabela. Sua principal função é registrar os dados da operação,

incluindo a data e hora em que o processo ocorreu, a quantidade processada durante o período de funcionamento e o número de carrinhos que circularam durante esse período. Além disso, essa tabela serve como chave estrangeira para referenciar as outras tabelas e identificar a operação associada a cada dado salvo. Isso permite filtrar e avaliar os dados de forma mais precisa.

- Tabela "info_carrinhos": Responsável por armazenar informações básicas sobre cada carrinho. Essas informações incluem o nome do carrinho, o horário de início da entrega, a duração da entrega e uma chave para filtrar por operação.

- Tabela "info_dados": Responsável por armazenar a evolução e resposta dos controles na aplicação. Os dados registrados nessa tabela incluem o nível do reservatório, a quantidade de carrinhos circulando na via (carrinhos em operação), o fluxo de carrinhos e o valor definido para o tempo de envio de cada carrinho no modo automático. É importante ressaltar que, quando o envio é feito manualmente, o valor registrado será 0. Além disso, essa tabela também possui uma chave para filtrar por operação.

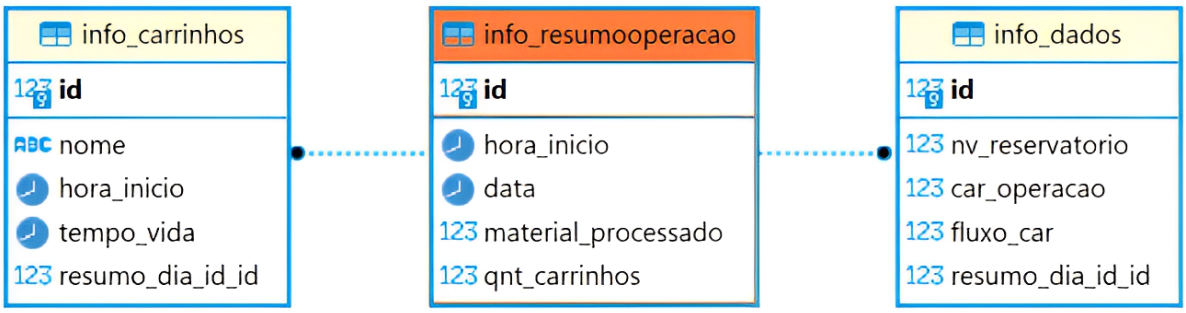


Figura 6 – Modelo do banco de dados.

Fonte: Autor.

3.3.3 Rota realtime

Para o envio de informações em tempo real, foi criada uma aplicação responsável unicamente por esta ação no projeto *Django*. Essa aplicação é composta por quatro arquivos:

- *"routing.py"*: responsável por indicar o caminho utilizado para a conexão com o *WebSocket*.

- *"urls.py"*: responsável por definir as rotas web disponíveis para essa aplicação.

- *"views.py"*: responsável por chamar o *template* e repassar informações importantes referentes a cada página.
- *"consumers.py"*: responsável por intermediar todo o processo envolvendo o *WebSocket* e apresenta as seguintes funções:
  - *"connect(self)"*: é chamada quando um cliente *WebSocket* se conecta. Ela extrai o nome da sala da rota URL e cria um nome de grupo para a sala. Em seguida, adiciona o canal do cliente ao grupo da sala e aceita a conexão.
  - *"disconnect(self, code)"*: é chamada quando um cliente *WebSocket* é desconectado. Ela remove o canal do cliente do grupo da sala.
  - *"receive(self, text_data)"*: é chamada quando o servidor recebe uma mensagem *WebSocket* do cliente. Converte os dados de texto em formato *JSON* e extrai a mensagem. Em seguida, envia a mensagem para o grupo da sala.
  - *"chat_message(self, event)"*: é chamada quando o servidor recebe uma mensagem do grupo da sala. Extrai a mensagem do evento recebido e a envia de volta para o cliente *WebSocket*.

Quando a rota HTTP é acionada, o correspondente *template* é carregado, o qual possui uma estrutura HTML e um *script* em *JavaScript* que gerencia todo o processo de envio e recebimento de mensagens no *front-end*. Assim, as informações coletadas pelo *routing* são enviadas para a página, onde são organizadas de acordo com as regras definidas no *script*.

3.3.4 Rotas CRUD

Um CRUD (*Create, Read, Update, Delete*) é um conjunto básico de operações amplamente utilizado em sistemas de gerenciamento de banco de dados e aplicações web. Ele representa as principais ações que podem ser realizadas em relação aos dados armazenados.

No projeto *Django*, foi desenvolvida uma aplicação adicional para acessar e modificar os dados disponíveis no PostgreSQL. Essa aplicação é responsável por gerar as rotas REST e retornar dados conforme solicitado. Para simplificar o processo, foi criada uma classe correspondente a cada tabela disponível no arquivo *"views.py"*. Essas classes herdam o método *"ModelViewSet"*. Para cada classe, um *"queryset"* e um *"serializer"* foram adicionados. O *"queryset"* é responsável por coletar as informações no banco de dados, enquanto o *"serializer"* converte essas informações em JSON para permitir a transferência de dados entre sistemas distintos. Essa ação por si só já disponibiliza via *Django* um conjunto de métodos CRUD para cada classe, além de uma consulta específica para cada tabela por meio do ID da informação.

Para oferecer mais opções aos usuários, foi criada uma função nas classes *"DadosViewSet"* e *"CarrinhosViewSet"*, que filtra os carrinhos de acordo com a operação. Isso permite que o usuário busque dados específicos. Por exemplo, se o usuário chamar a rota ".../resumo_operacao/", que é controlada pela *view "ResumoOperacaoViewSet"*, receberá todas as operações realizadas até o momento. O usuário pode verificar as operações e escolher uma que deseja ver mais detalhes, anotar o ID e acessar a rota ".../carrinhos/get_info_carrinhos_por_operacao/id/" passando o ID escolhido. Dessa forma, apenas os dados dos carrinhos daquela operação específica serão retornados.

Para acessar essas rotas, o usuário pode utilizar um navegador qualquer, chamando diretamente o servidor, pois o *Django* já disponibiliza uma interface que permite a realização de solicitações HTTP. Além disso, é possível utilizar programas como o *Postman* e o *Insomnia* para fazer as requisições HTTP, fornecendo a URL e o verbo HTTP desejado.

Como esse projeto não será colocado em produção, as rotas foram mantidas abertas, o que significa que qualquer pessoa que as acesse poderá visualizar e modificar os dados. No entanto, é possível limitar o acesso a usuários específicos, aplicar regras de autenticação e bloquear determinados métodos, como o *Delete*. No entanto, como esse não era o foco do projeto, as rotas foram mantidas disponíveis para todos os usuários, independentemente de estarem autenticados/logados ou não.

3.3.5 Rota Gráficos

Para enriquecer ainda mais o trabalho, foi implementada uma funcionalidade adicional na aplicação responsável pelo CRUD. Foi desenvolvida uma rota chamada *"chart"* que recebe o ID de uma operação específica como parâmetro. Essa rota retorna dois gráficos que exibem os dados coletados durante o período de funcionamento.

No arquivo *"view.py"*, foi criada uma função dedicada a essa rota chamada *"chart_view"*. Essa função é responsável por buscar os dados necessários e repassá-los a um *template* específico. Esse *template* é responsável por coletar os dados e plotar as informações em uma página da web. Para realizar a plotagem dos gráficos, é utilizado um *script* em *JavaScript* que faz uso da biblioteca *Canvas*.

Além disso, a página gerada possui barras de intervalo que permitem ao usuário limitar um início e um fim para visualizar apenas uma parte selecionada dos dados de forma mais detalhada. Isso proporciona uma análise mais específica e focada nas informações desejadas.

4 Resultados e Discussão

Após a conclusão de todas as etapas, foi possível alcançar o fluxograma de funcionamento representado pela Figura 7, que ilustra o seguinte processo: o *C#* desempenha todas as operações necessárias e alimenta o banco de dados PostgreSQL. Para efetuar a publicação na rota *realtime*, ele busca autorização junto ao *Django*, responsável por executar todas as ações requeridas para permitir um acesso adequado ao *WebSocket* e compartilhar as mensagens pertinentes. O *Django*, por sua vez, tem a capacidade de receber e manipular os dados do banco de dados, organizando-os de forma a retornar as informações solicitadas pelas respectivas rotas. Além disso, o *Django* exerce total controle sobre a conexão e a correta transmissão de mensagens por parte dos clientes do *WebSocke*t.

O mencionado fluxograma evidencia de maneira clara as responsabilidades atribuídas a cada componente envolvido: cabe ao *C#* o controle, o funcionamento, a interface e a geração de dados, ao passo que ao *Django* é incumbida a tarefa de assegurar o acesso e o repasse de informações relevantes por meio da Web.

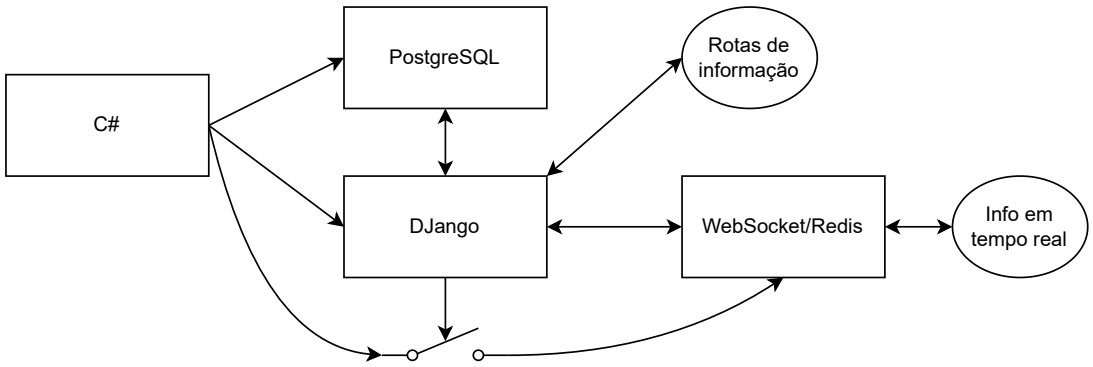


Figura 7 – Fluxograma do projeto.

Fonte: Autor.

Em sua forma final, a interface assume a aparência ilustrada na Figura 8, destacando-se a presença de dados em constante atualização, os botões de controle previamente mencionados no capítulo anterior, uma representação animada em tempo real da estrada percorrida pelos carrinhos, a animação correspondente à reserva do alto-forno e, por último, uma *listbox* contendo informações específicas referentes a cada carrinho de entrega.

Na figura, é possível observar o funcionamento da estratégia de não colisão, em que os carrinhos em percurso mantêm a distância pré-estabelecida durante o planejamento do projeto. No cruzamento, apenas um carrinho o ocupa por vez, seguindo a lógica do "*Semaphore*" com apenas uma vaga disponível. Portanto, enquanto um carrinho não

terminar de atravessar, nenhum outro pode adentrar naquele espaço.

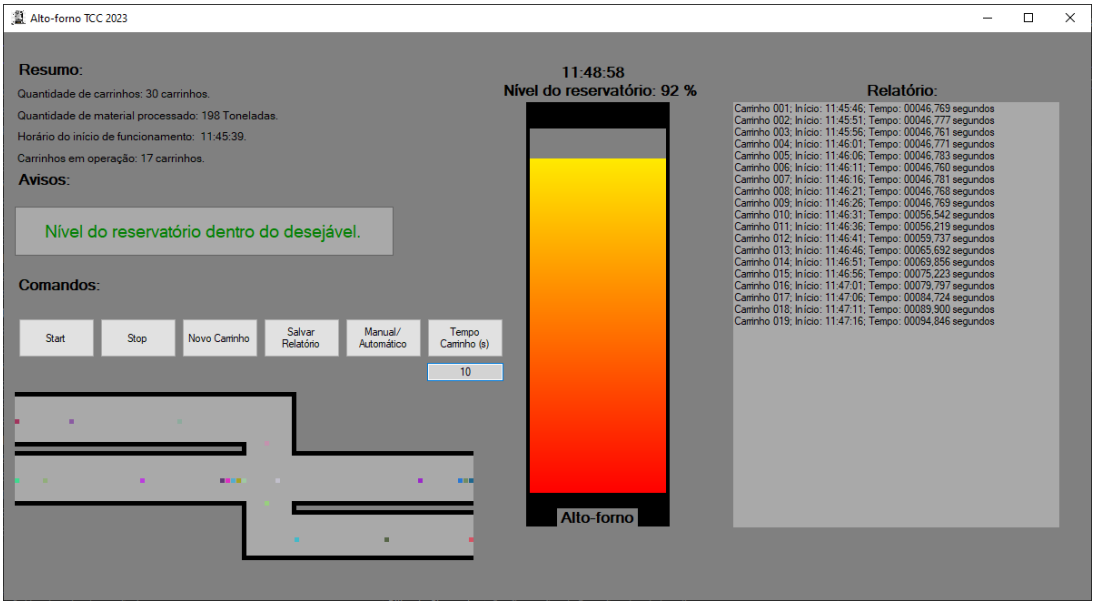


Figura 8 – Layout de controle do projeto finalizado.

Fonte: Autor.

Na Figura 9, podemos observar a resposta ao acessar a rota "*resumo_operacao*" através do navegador. Essa interface é gerada e disponibilizada pelo DRF, fornecendo toda a funcionalidade necessária para operações via rotas RESTful. Na Figura 10, é demonstrado o uso da mesma rota, porém utilizando o *Postman*. É importante notar que, para o ID 32, a rota retorna exatamente as mesmas mensagens no método GET, tanto no *Postman* quanto pela API, o que evidencia a correta funcionalidade da rota. Vale ressaltar que, nos dois casos, se nenhum valor de ID fosse fornecido, a rota retornaria como resposta todos os valores disponíveis na tabela "*ResumoOperacao*".

Na Figura 11, é possível observar a apresentação das informações em tempo real. É importante ressaltar que não há retenção de mensagens, portanto, todos os clientes que acessarem esse serviço receberão a mesma mensagem simultaneamente, mas apenas aquelas que forem enviadas após o acesso do cliente. Para visualizar dados anteriores ao seu acesso, o cliente pode utilizar consultas através das URLs REST disponíveis, como já demonstrado anteriormente. Além disso, a página web possui uma caixa de digitação e um botão de envio, o que permite ao cliente enviar mensagens para todos os clientes conectados neste servidor.

A fim de avaliar o desempenho do projeto em estudo, consideremos a Figura 12 como um exemplo representativo. Essa figura foi gerada através dos dados coletados pela execução local do projeto por aproximadamente 3 horas e engloba as informações essenciais sobre o processo, incluindo um gráfico que representa a variação do nível do reservatório

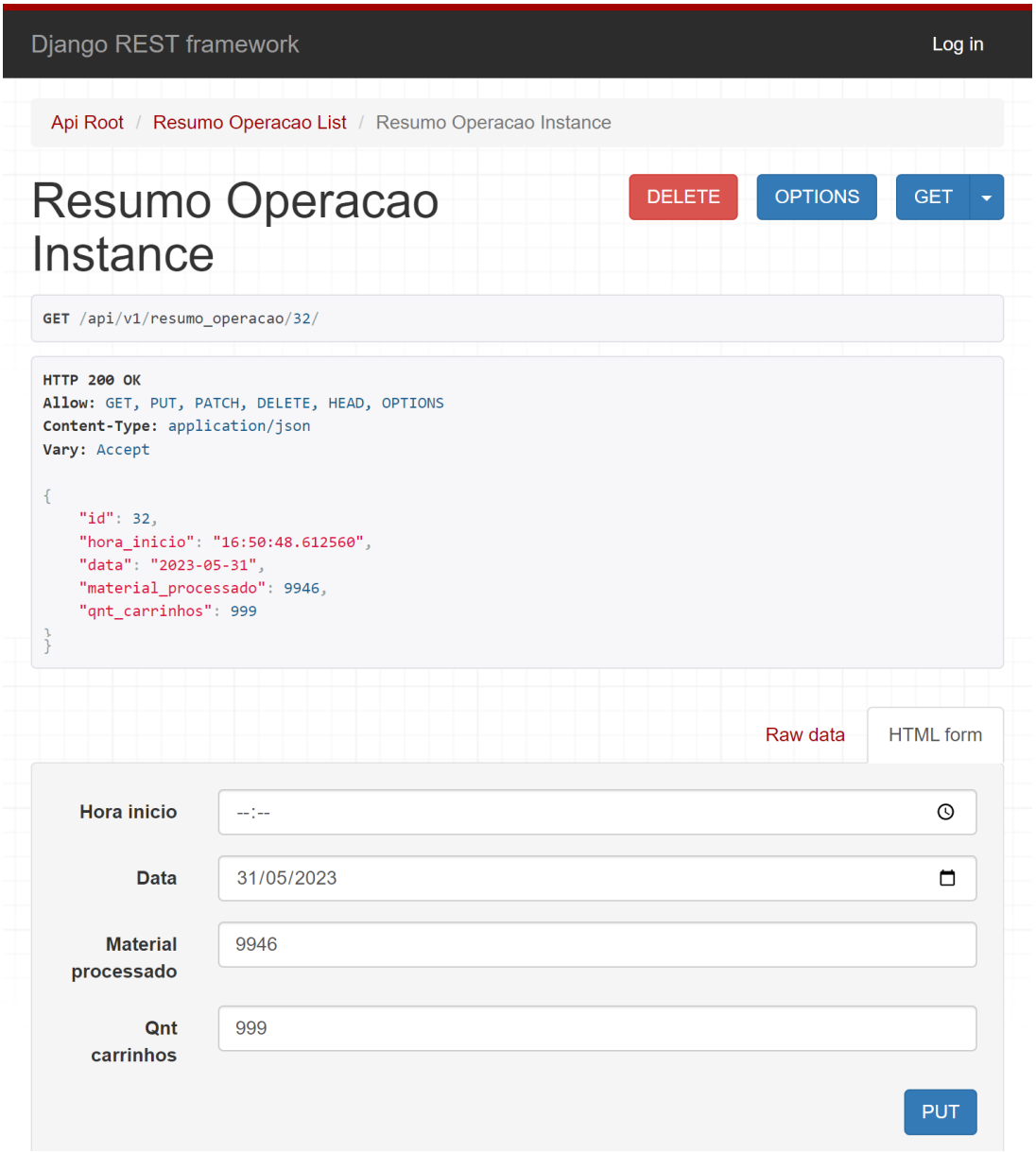


Figura 9 – Rota disponibilizada pelo Django para solicitações HTTP RESTful

Fonte: Autor.

ao longo do tempo, mostrado em azul. Em vermelho, temos a quantidade de carrinhos em circulação em momentos específicos do processo, e em verde, o fluxo dos carrinhos, ou seja, a frequência de envio dos mesmos.

No início das medições, observamos que o alto-forno iniciou o processo com o reservatório cheio. Entretanto, devido a um atraso na medição, o primeiro valor registrado está próximo de 90%. Nesse estágio inicial, é notável que o fluxo de carrinhos foi ajustado para enviar um carrinho a cada segundo, o que afeta diretamente a quantidade de carrinhos em circulação, conforme evidenciado pela ascensão do gráfico de fluxo em vermelho. Durante esse intervalo, o nível do reservatório diminui, pois os insumos ainda estão a caminho.

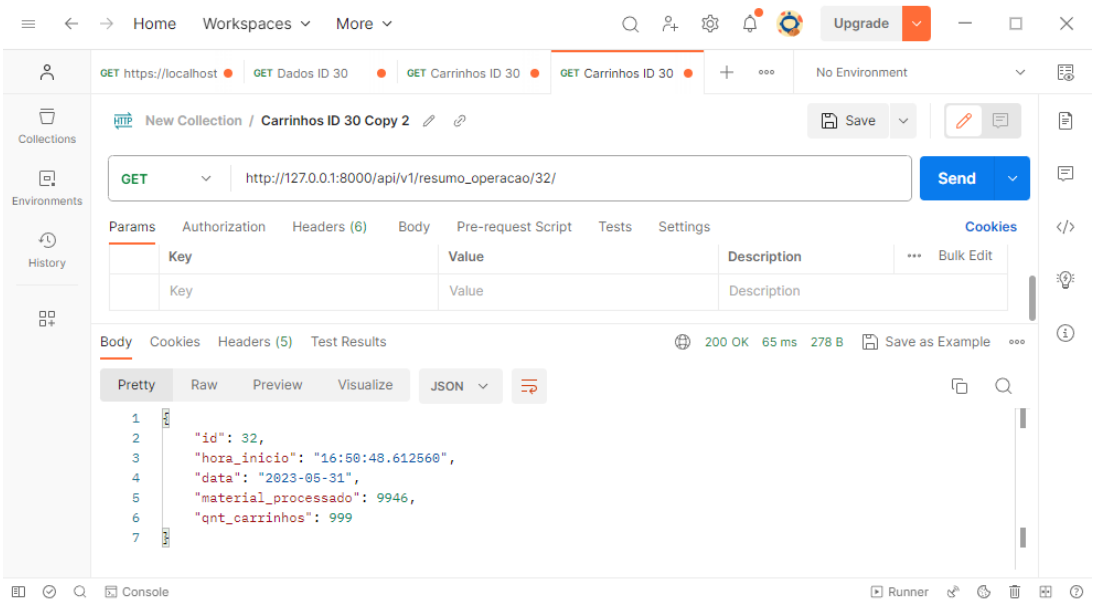


Figura 10 – Demonstração do funcionamento das rotas por outro métodos como o Postman.

Fonte: Autor.

Portanto, o alto-forno está consumindo sua reserva sem receber nenhum material.

Após essa fase, à medida que o nível do reservatório começa a aumentar, a frequência de envio é reduzida para 20 e, posteriormente, para 10, resultando na estabilização do nível do reservatório entre 90% e 100%. Ao mesmo tempo, o fluxo de envio de carrinhos ocorre a cada 10 segundos, e a quantidade de carrinhos em operação se aproxima de 13.

Com o intuito de simular uma escassez de material, o fornecimento é interrompido por volta dos 28 minutos. É importante observar que há um atraso na queda do reservatório devido à presença dos carrinhos em circulação. Após aproximadamente 4 minutos, não há mais carrinhos em circulação, e o reservatório do alto-forno esvazia-se, sem a emissão de novos carrinhos. Para retomar a operação, por volta do minuto 34, o envio de insumos é reiniciado, resultando em uma situação de sobrecarga. É possível observar um pico de mais de 60 carrinhos transitando simultaneamente próximo ao minuto 48. Posteriormente, alguns ajustes no envio são realizados, e o sistema entra em estabilidade, mantendo-se assim até o final do processo, Figura 13.

No segundo gráfico disponível na imagem, é possível observar o tempo necessário para cada carrinho completar a tarefa de entrega. É notável que, durante o período de congestionamento apresentado no gráfico anterior, entre os minutos 35 e 59, houve um aumento significativo no tempo de execução, atingindo um tempo máximo de espera superior a 8 minutos. Esse valor é 10 vezes maior do que o necessário para uma entrega durante a fase de estabilidade no final do processo. Importante destacar que, mesmo em sobrecarga, o alto-forno demonstrou um desempenho estável e robusto, mantendo seu nível

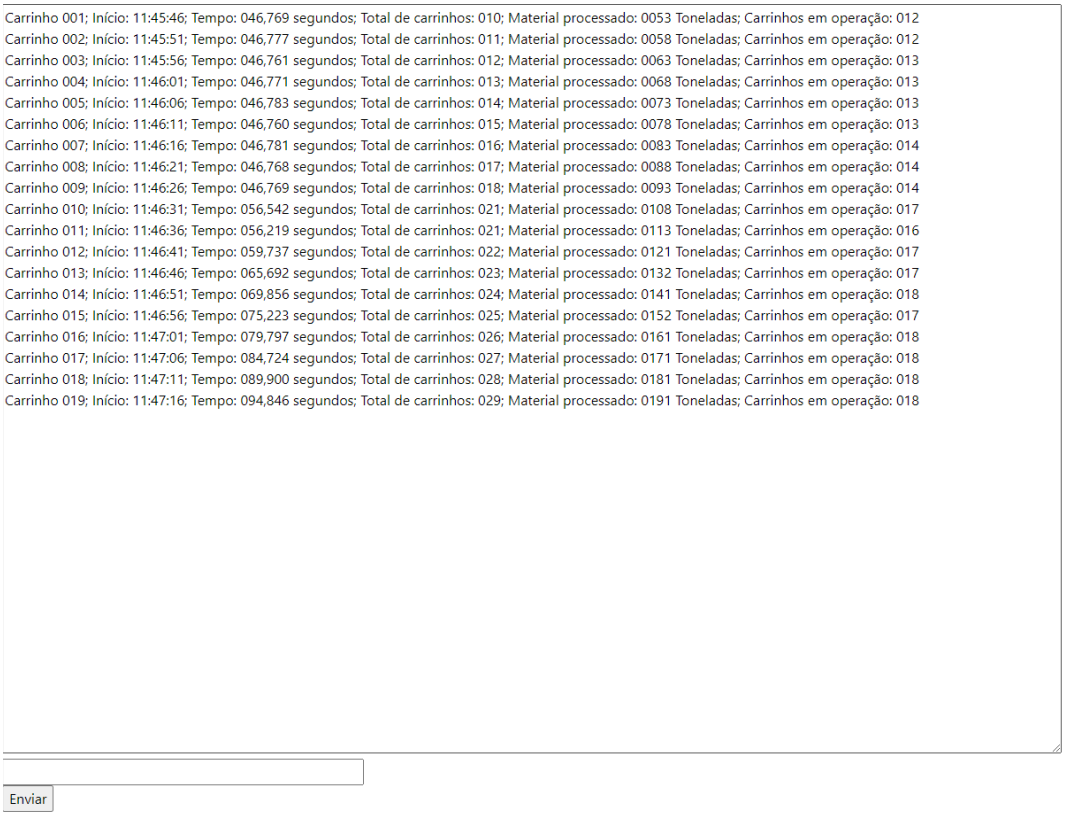


Figura 11 – Rota que recebe as informações em tempo real gerados pelo programa principal.

Fonte: Autor.

de reservatório sempre entre 90% e 100%.

Por fim, graças ao *Swagger*, o projeto possui uma página de documentação das rotas REST disponíveis (Figuras 14 e 15). Nessas figuras, é possível visualizar o seu design ao ser acessado através de um navegador. As rotas disponíveis com seus respectivos verbos HTTP são apresentadas, juntamente com uma aba expansível que fornece detalhes sobre cada rota. Além disso, a documentação gerada também inclui os modelos disponíveis no banco de dados, bem como os detalhes expandidos de cada tabela, quando expostos. É importante ressaltar que caso as rotas disponíveis na interface do Swagger sejam acessadas em um navegador, diretamente pela API, todas elas retornarão uma interface semelhante à apresentada na Figura 9.

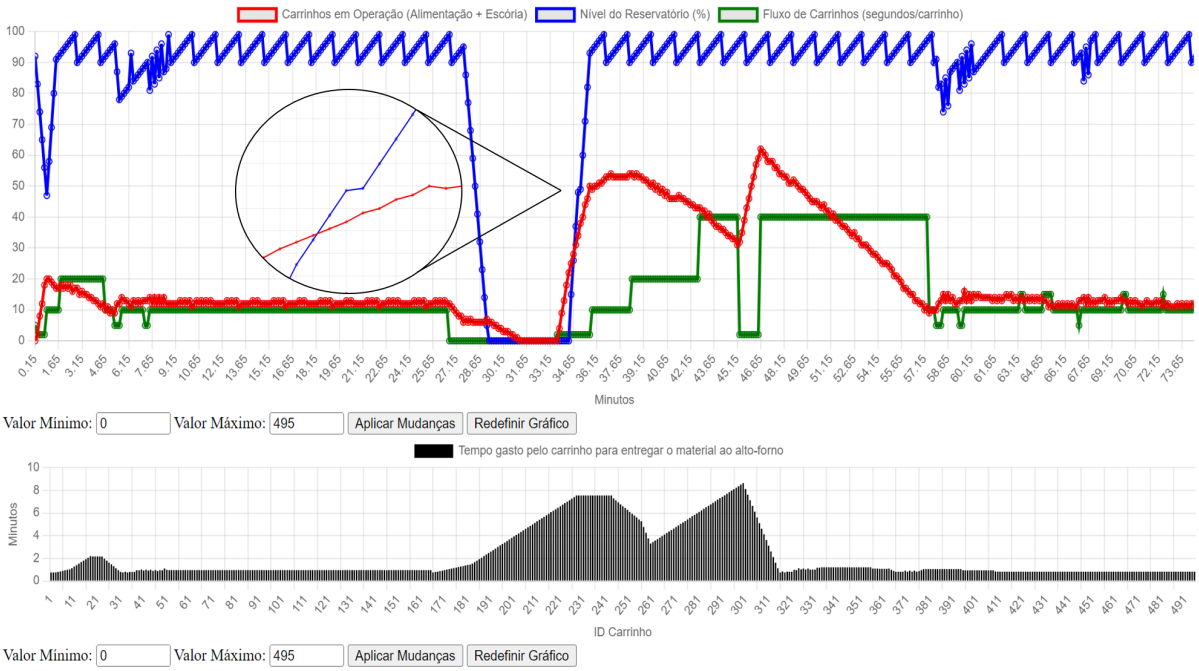


Figura 12 – Rota *chart* do projeto com os dados da operação de ID 32, parte 1.

Fonte: Autor.

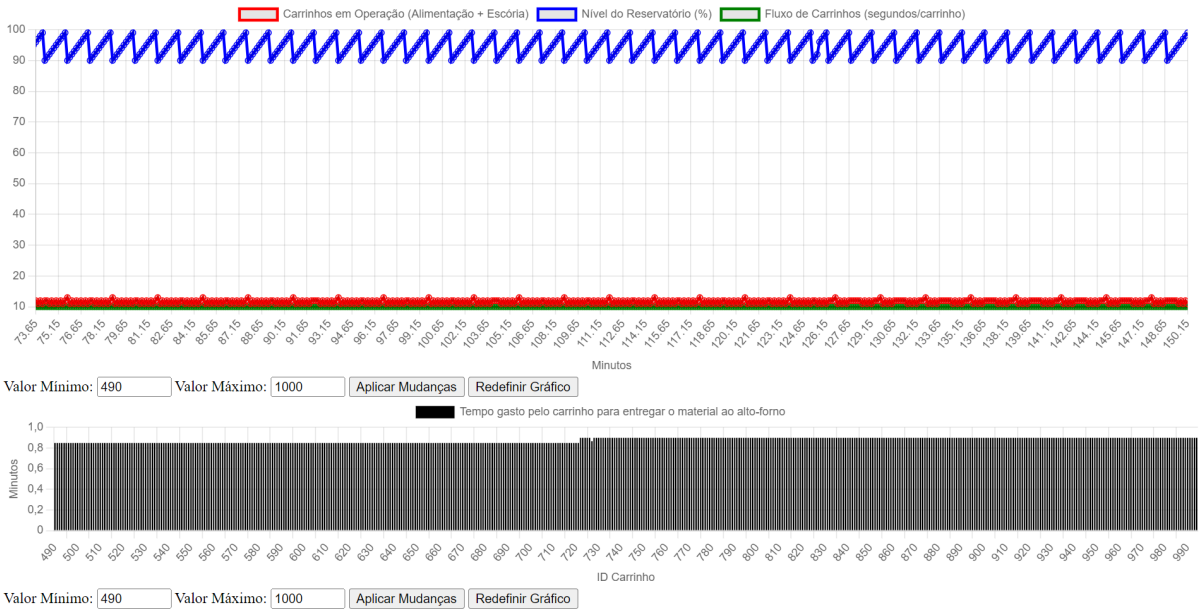


Figura 13 – Rota *chart* do projeto com os dados da operação de ID 32, , parte 2.

Fonte: Autor.

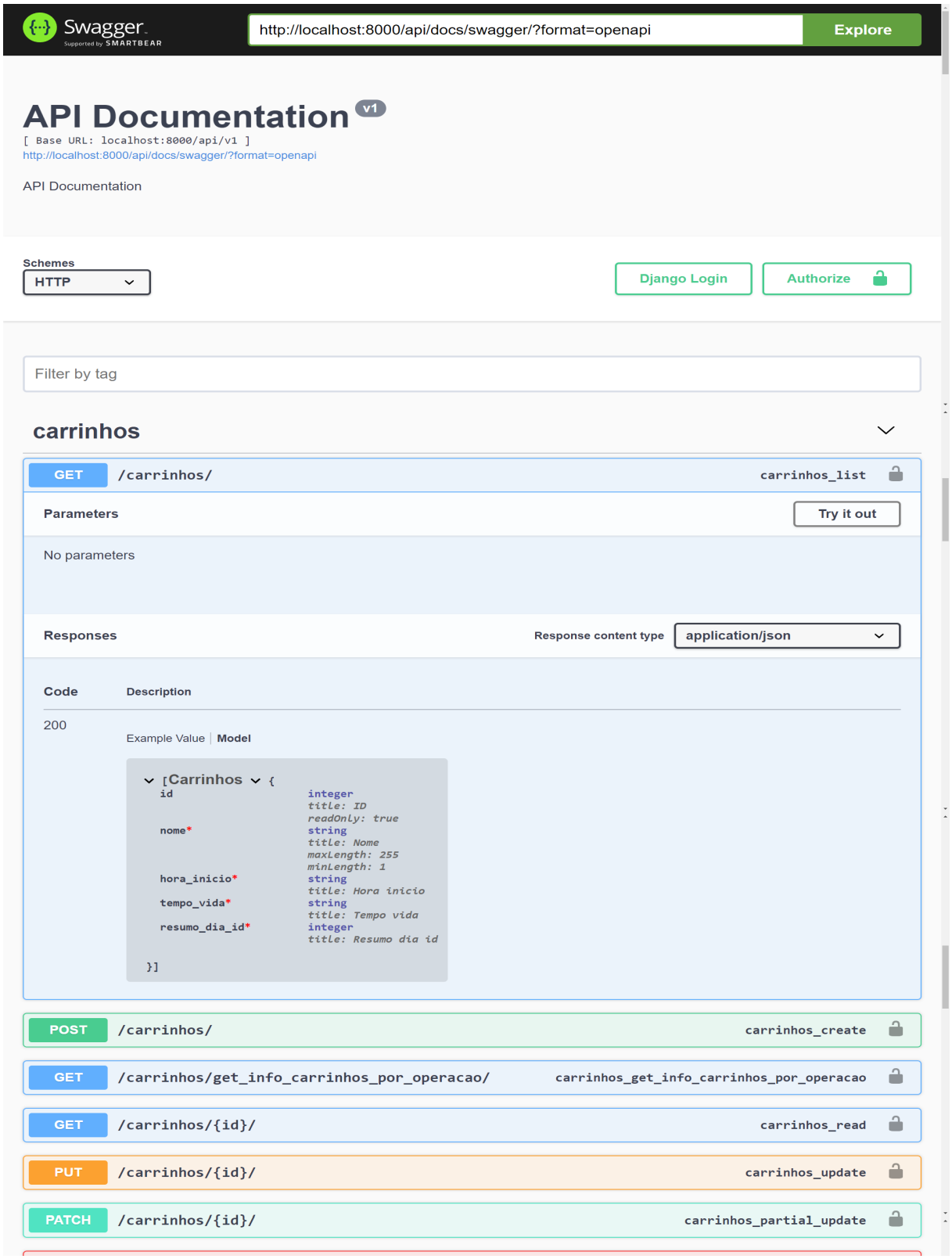


Figura 14 – Documentação feita pelo *Swagger* das rotas e modelos existentes no projeto, parte 1.

Fonte: Autor.

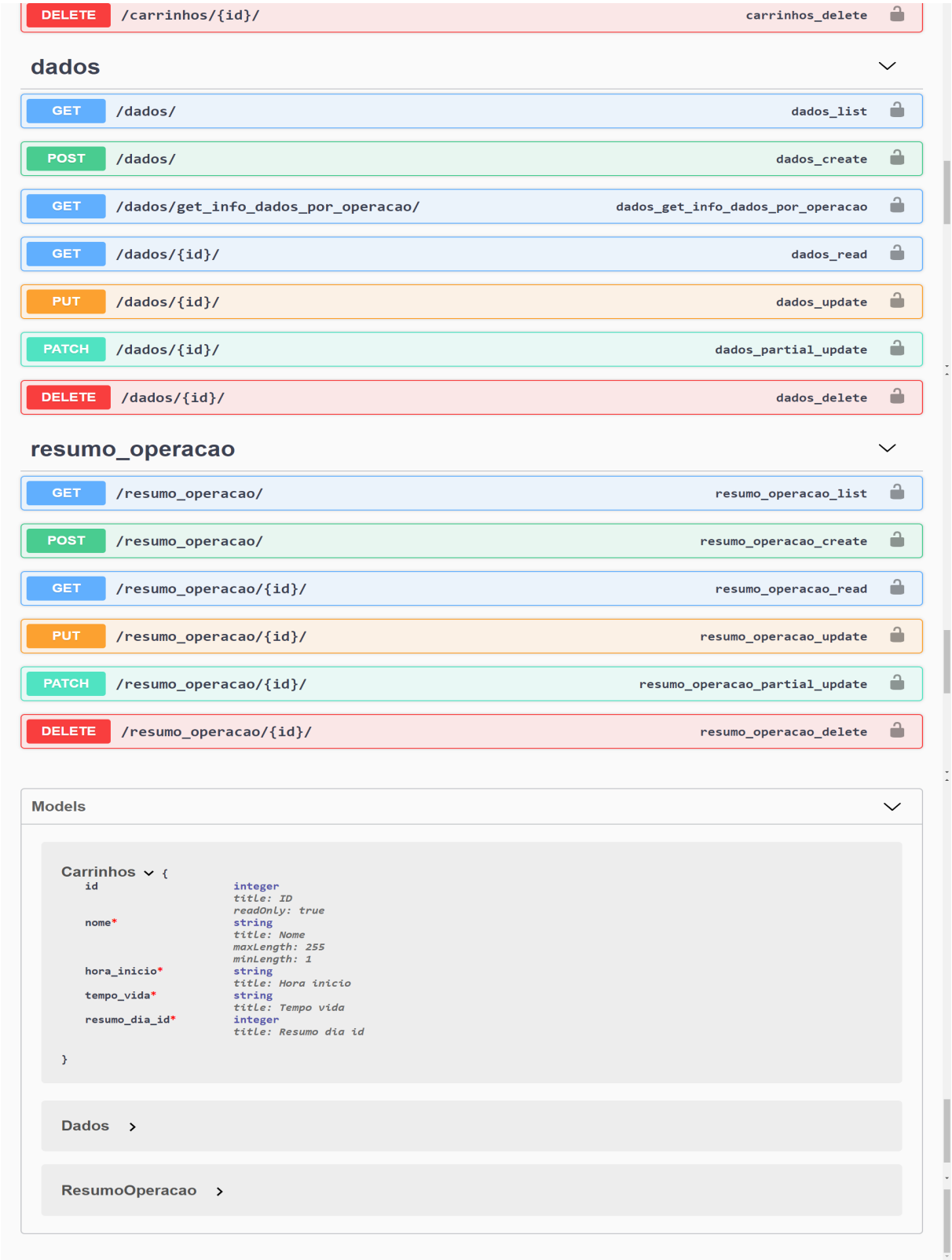


Figura 15 – Documentação feita pelo *Swagger* das rotas e modelos existentes no projeto, parte 2.

Fonte: Autor.

5 Considerações Finais

Este trabalho abordou um sistema de abastecimento de um alto-forno, no qual carrinhos com material percorrem caminhos para transportar suas respectivas cargas. Como desafio do projeto, as estradas de escória e insumos se encontravam em um cruzamento com uma regra limitante de trânsito de apenas um veículo por vez. Além disso, foram desenvolvidos com sucesso controles de operação do sistema, nos quais todos responderam conforme o esperado. Adicionalmente, o armazenamento de dados mostrou-se eficiente e funcionou permitindo análises completas sobre o funcionamento do sistema.

O sistema de rotas RESTful utilizando o *Django* desempenhou seu papel ao permitir a obtenção dos dados de operação via web para qualquer usuário. O sistema de envio de informações em tempo real via *WebSocket* também cumpriu sua função ao enviar os dados para todos os clientes inscritos na rota. A rota "*chart*", responsável por fornecer gráficos com dados da operação, está funcional, e seus recursos de zoom estão respondendo conforme desejado.

Ao analisar o gráfico de uma operação específica, foi possível observar reflexos da operação, como a espera do carrinho de insumo para que o carrinho de escória termine de atravessar a via, o que resultou na perturbação da reta de ascensão do enchimento do alto-forno. Além disso, foi possível verificar a resposta do sistema a cenários críticos, como engarrafamento de veículos e falta de material, nos quais o sistema reagiu conforme o esperado, apresentando estabilidade nos momentos de grande volume de tráfego e resposta rápida nos momentos de escassez. Quanto a respostas em cenários controlados e estáveis, o tempo médio para entrega de material foi de 51 segundos, enquanto o tempo máximo foi de 8 minutos e 39 segundos para cenários de engarrafamento. Na operação avaliada, o sistema funcionou sem nenhum problema desde a ausência de carrinhos até o pico de 62 carrinhos simultâneos. Em relação à estabilidade apresentada no restante final do processo avaliado, a média de carrinhos circulando (insumos mais escória) foi de 12 carrinhos simultâneos.

O projeto foi estruturado com 999 carrinhos de entrega de insumos disponíveis que não retornam. Assim, quando esse valor é alcançado, o sistema é encerrado, o que pode ser considerado um *bug*. Uma alteração futura poderia ser tornar o caminho cíclico, de forma que seja possível diminuir a quantidade de carrinhos, mas torná-los funcionais por tempo indeterminado.

Apesar de não ser o foco do projeto, toda a estrutura desenvolvida abre possibilidades para diversas melhorias, como a implementação de um sistema administrativo para acesso ao banco de dados, o desenvolvimento de novas formas de coleta de dados e a introdução de novos recursos Web. Além disso, é possível levar o sistema à produção,

hospedando-o em provedores como *AWS* e *Azure*, o que permitiria o acesso real via web a todas as informações geradas.

Assim, este trabalho foi bem-sucedido em cumprir todos os desafios propostos, desenvolvendo um software de simulação robusto e estável de um processo de automação interativo, com a possibilidade de acesso à consultas de informações via Web tanto de formas numéricas, textuais e gráficas.

Referências

AWS. *O que é SQL?* 2023. <https://aws.amazon.com/pt/what-is/sql/>. Accessed: 2023-06-12. Citado na página 19.

AZEVEDO, I. B. de. *O prazer da producao cientifica / diretrizes para elaboracao de trabalhos academicos*. 5. ed. Piracicaba, SP: UNIMEP, 1997. Citado na página 14.

AZURE. *O que são os bancos de dados NoSQL?* 2023. <https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-nosql-database>. Accessed: 2023-06-12. Citado na página 20.

FERNANDES, E. A. *ProjetoTCC*. 2023. <https://github.com/ErickFernan/TCC_AltoForno_Django>. Accessed: 2023-06-19. Citado na página 25.

FERNANDES, E. A. *Requirements.txt*. 2023. <https://github.com/ErickFernan/TCC_AltoForno_Django/blob/main/requirements.txt>. Accessed: 2023-06-19. Citado na página 30.

FORCIER, J.; BISSEX, P.; CHUN, W. *Python Web Development with Django*. 1. ed. [S.l.]: Addison-Wesley Professional, 2008. ISBN 0132356139. Citado na página 23.

HOLOVATY, A.; KAPLAN-MOSS, J. *The Definitive Guide to Django: Web Development Done Right, Second Edition*. 2nd. ed. USA: Apress, 2009. ISBN 143021936X. Citado na página 22.

INFOMET. *Aços Ligas | Aço: Processos de Fabricação | Processo Siderúrgico*. 1998. <https://www.infomet.com.br/site/acos-e-ligas-conteudo-ler.php?codConteudo=234>. Accessed: 2023-06-10. Citado 2 vezes nas páginas 12 e 15.

INSIGHTLAB. *Por que o Python é a Linguagem mais adotada na área de Data Science ?* 2019. <https://www.insightlab.ufc.br/por-que-o-python-e-a-linguagem-mais-adotada-na-area-de-data-science/>. Accessed: 2023-06-10. Citado na página 16.

INSTITUTE, P. *Python® – the language of today and tomorrow*. 2012. <https://pythoninstitute.org/about-python>. Accessed: 2023-06-10. Citado na página 16.

MDN contributors. *DOM (Document Object Model)*. 2023. <https://developer.mozilla.org/en-US/docs/Glossary/DOM>. Accessed: 2023-06-10. Citado na página 16.

MDN contributors. *JavaScript basics*. 2023. <https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics>. Accessed: 2023-06-10. Citado na página 16.

MELNIKOV, A.; FETTE, I. *The WebSocket Protocol*. RFC Editor, 2011. RFC 6455. (Request for Comments, 6455). Disponível em: <https://www.rfc-editor.org/info/rfc6455>. Citado na página 21.

MICROSOFT. *Mutexes*. 2023. <https://learn.microsoft.com/pt-br/dotnet/standard/threading/mutexes>. Accessed: 2023-06-12. Citado na página 18.

MICROSOFT. *Perguntas frequentes sobre o Subsistema Windows para Linux*. 2023. <https://learn.microsoft.com/pt-br/windows/wsl/faq>. Accessed: 2023-06-13. Citado na página 23.

MICROSOFT. *Semaphore e SemaphoreSlim*. 2023. <https://learn.microsoft.com/pt-br/dotnet/standard/threading/semaphore-and-semaphoreslim>. Accessed: 2023-06-12. Citado na página 18.

MICROSOFT. *Threads e threading*. 2023. <https://learn.microsoft.com/pt-br/dotnet/standard/threading/threads-and-threading>. Accessed: 2023-06-11. Citado 2 vezes nas páginas 17 e 18.

MICROSOFT. *Um tour pela linguagem C*. 2023. <https://learn.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/>. Accessed: 2023-06-10. Citado na página 15.

MINERAL, C. *Após 34 meses desativado e uma grande reforma, alto-forno da Usiminas é reativado*. 2018. <https://www.conexaomineral.com.br/noticia/984/apos-34-meses-desativado-e-uma-grande-reforma-alto-forno-da-usiminas-e-reativado.html>. Accessed: 2023-06-10. Citado na página 14.

MONITOR dotcom. *Monitoramento de aplicativos do WebSocket*. 2013. <https://www.dotcom-monitor.com/blog/pt-br/monitoramento-de-aplicativos-do-websocket/>. Accessed: 2023-06-13. Citado na página 22.

MONITOR dotcom. *Install Redis on Windows*. 2023. <https://redis.io/docs/getting-started/installation/install-redis-on-windows/>. Accessed: 2023-06-13. Citado na página 23.

OLIVEIRA, R. de. *Fundamentos Dos Sistemas de Tempo Real*. Independently Published, 2018. ISBN 9781728694047. Disponível em: <https://books.google.com.br/books?id=I2umyQEACAAJ>. Citado na página 12.

PETERS, T. *The Zen of Python*. 2004. <https://pep20.org/>. Accessed: 2023-06-10. Citado na página 16.

POSTGRESQL. *About*. 2023. <https://www.postgresql.org/about/>. Accessed: 2023-06-12. Citado na página 20.

REDIS. *Introduction to Redis*. 2023. <https://redis.io/docs/about/>. Accessed: 2023-06-12. Citado na página 20.

REYNOLDS, J. *8 World-Class Software Companies That Use Python*. 2018. <https://realpython.com/world-class-companies-using-python/>. Accessed: 2023-06-10. Citado na página 16.

RIZZO, E. M. da S. *Processo de fabricação de ferro-gusa em alto-forno*. 1. ed. São Paulo, SP: ABM, 2009. Citado 2 vezes nas páginas 12 e 15.

THAKUR, S. *Know The Difference Between Mutex Semaphore In Operating System*. 2022. <https://unstop.com/blog/difference-between-mutex-and-semaphore>. Accessed: 2023-06-12. Citado 2 vezes nas páginas 8 e 19.