Gestão, Tecnologia e Inovação
Revista eletrônica dos Cursos de Engenharia

Desenvolvimento de um Sistema Multiplataforma para Gerenciamento de Ordens de Serviço

Gustavo Walk Gavronski, gustawalk@gmail.com, Universidade UniOpet

RESUMO

O presente artigo propõe o desenvolvimento de um sistema multiplataforma leve para o gerenciamento de ordens de serviço em empresas de assistência técnica. Apresentando a relevância das ordens de serviço como documentos formais de controle das atividades de manutenção, destacando sua importância para organização e comunicação entre técnicos e gestores. Descrevemos objetivos e motivação do projeto, incluindo a necessidade de mobilidade e operação offline que soluções existentes não atendem adequadamente. A metodologia de engenharia de software adotada envolve levantamento de requisitos, modelagem e implementação iterativa usando as tecnologias Tauri, TypeScript e Rust. Os resultados esperados incluem automação de tarefas administrativas, geração automática de contratos em PDF e melhoria na eficiência operacional. Por fim, concluímos que a solução proposta oferece uma alternativa moderna e acessível para empresas do setor, combinando flexibilidade híbrida de banco de dados (MariaDB local/SQlLite) com uma interface amigável.

PALAVRAS CHAVES

gerenciamento de ordens de serviço; sistema multiplataforma; manutenção industrial; Tauri; Rust.

1 INTRODUÇÃO

As ordens de serviço (OS) são documentos formais que detalham as tarefas e intervenções técnicas a serem executadas, padronizando o fluxo de trabalho em ambientes de manutenção industrial. Segundo a IBM (2023), a OS é um documento formal que contém “todos os detalhes das tarefas de manutenção e descreve um processo para concluí-las”. Isso assegura clareza na comunicação entre cliente, equipe técnica e gestão, mantendo um registro histórico das intervenções realizadas. Em empresas de assistência técnica, a organização eficiente dessas ordens é fundamental para evitar atrasos e retrabalho, pois alinha responsabilidades e prazos entre os envolvidos.

Neste contexto, torna-se essencial criar um sistema de informação que gerencie essas ordens de serviço de forma integrada. O objetivo deste trabalho é propor uma solução de software multiplataforma e leve, capaz de operar também em modo offline. Propomos uma arquitetura em camadas (front-end, back-end e banco de dados) utilizando tecnologias modernas como o framework Tauri (que integra front-end web com lógica em Rust), TypeScript e Rust. Tal sistema busca solucionar problemas recorrentes em plataformas existentes, como dependência constante de conexão à internet e controle manual de documentos, além de oferecer funcionalidades adicionais como geração automática de contratos em PDF.

Este projeto visa, portanto, desenvolver uma aplicação eficiente, moderna e acessível, que possibilite às empresas de assistência técnica centralizar informações de ordens de serviço e contratos em um dashboard integrado. Espera-se que o sistema contribua para padronizar processos, reduzir erros operacionais e aumentar a produtividade da equipe, conforme já ressaltado na literatura de melhoria de processos organizacionais. A seguir, apresentamos a fundamentação teórica que embasa esse desenvolvimento, detalhando conceitos de ordem de serviço, princípios de engenharia de software, requisitos, arquitetura de software, banco de dados e o framework Tauri.

2 REVISAO DA LITERATURA

2.1 ORDEM DE SERVICO

Uma ordem de serviço (OS) formaliza e define o trabalho contratado, detalhando todas as informações sobre o serviço a ser prestado. Ela descreve de forma precisa o escopo do trabalho, garantindo que cliente, fornecedores e equipe técnica estejam cientes do conteúdo do serviço. Na prática, a OS serve como referência para evitar dúvidas posteriores sobre o que foi acordado, listando atividades, materiais necessários e responsáveis. Conforme destacado pela IBM (2023), a OS é a força motriz das operações de manutenção, mantendo todas as partes informadas sobre o fluxo de trabalho. Em empresas técnicas, a emissão de ordens de serviço assegura clareza operacional e jurídica, orientando o colaborador sobre seu trabalho e fornecendo ao cliente a confirmação dos serviços contratados.

2.2 ENGENHARIA DE SOFTWARE

A engenharia de software orienta o desenvolvimento do sistema por meio de processos e práticas reconhecidas. Seu objetivo é planejar e gerenciar recursos de desenvolvimento para obter um software de qualidade dentro dos custos planejados. Neste projeto, adotaremos um processo iterativo e incremental baseado em metodologias ágeis (por exemplo, Scrum), permitindo entregas contínuas de funcionalidades em ciclos curtos. Essa abordagem ágil promove feedback frequente dos usuários e ajustes rápidos de requisitos, o que tende a reduzir erros e custos ao longo do desenvolvimento.

Para garantir compatibilidade multiplataforma, o sistema empregará frameworks de desenvolvimento móvel/web que suportem múltiplos sistemas operacionais no front-end, combinados a serviços web RESTful para integração entre clientes e servidor. A arquitetura do software será organizada em camadas (apresentação, negócio e dados) para separar responsabilidades e facilitar manutenção. Além disso, adotamos práticas de qualidade como controle de versão (Git), testes automatizados (unitários e de integração) e revisão de código, assegurando robustez e facilidade de evolução do sistema.

2.3 ENGENHARIA DE REQUISITOS

A engenharia de requisitos envolve o processo disciplinado de elicitação, especificação, análise, validação e gerenciamento das necessidades do sistema. Segundo Pohl (1994), trata-se de “uma abordagem sistemática e disciplinada para a especificação e gerenciamento de requisitos”. Em consonância, Sommerville (2011) define requisito como “as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento”. Isso significa documentar o que o sistema precisa realizar sem detalhar como será implementado.

No projeto, os requisitos foram coletados por meio de técnicas combinadas de elicitação: entrevistas com usuários-chave, questionários e análise de documentos do processo atual. Essa abordagem múltipla é recomendada porque “na maioria dos projetos, mais de uma técnica e abordagem de elicitação de requisitos precisará ser usada” para obter melhores resultados. Desta forma, identificamos requisitos funcionais (como cadastro de clientes, criação de ordens de serviço, geração de contratos e consulta de status) e não funcionais (segurança, desempenho, usabilidade etc.). Os requisitos funcionais principais foram modelados em *diagramas de caso de uso* – diagramas UML que representam as interações entre atores (por exemplo, cliente, técnico e administrador) e o sistema, facilitando a compreensão do escopo do sistema. A especificação final dos requisitos foi organizada em documentos formais e *user stories* em um backlog, permitindo a revisão iterativa e priorização ao longo do desenvolvimento. Todos os requisitos foram validados continuamente com os stakeholders e refinados conforme o feedback, garantindo alinhamento com as necessidades reais do usuário.

2.4 SOFTWARES SIMILARES

Integramos ao referencial teórico uma análise de softwares similares para fundamentar o escopo do projeto. A análise da concorrência, ou benchmarking, consiste em investigar o que os concorrentes diretos estão fazendo para identificar melhores práticas. Em outras palavras, buscamos “identificar quais são as organizações que se destacam em algo em particular” e aprender com seus exemplos. No contexto de gestão de ordens de serviço, estudamos sistemas existentes do mesmo segmento (por exemplo, soluções comerciais de OS), mapeando funcionalidades comuns (cadastro de OS, emissão de contratos, acompanhamento de manutenção) e critérios de qualidade (fluxo de trabalho, interface). Esses insights orientaram o levantamento de requisitos e ajudaram a definir padrões e funcionalidades essenciais no projeto, evitando reinventar soluções já consolidadas e adotando práticas testadas no mercado.

2.5 ARQUITETURA DE SOFTWARE

A arquitetura de software estabelece a estrutura organizacional de um sistema, incluindo seus componentes principais, relacionamentos e diretrizes de design. Bass, Clements e Kazman (2012) definem arquitetura como “o conjunto de estruturas necessárias para raciocinar sobre o sistema”. Uma boa arquitetura proporciona flexibilidade, escalabilidade e manutenibilidade. Entre os estilos arquiteturais comuns estão monolítico, em camadas, cliente-servidor, orientado a serviços (SOA) e microsserviços. Neste projeto, adotamos uma arquitetura em camadas, separando: interface do usuário (front-end em TypeScript), lógica de aplicação (back-end em Rust via Tauri) e persistência de dados (banco MariaDB remoto e SQLite local). Essa divisão modular facilita a organização do código, a manutenção do sistema e futuras adaptações para diferentes plataformas, permitindo que cada camada evolua independentemente.

2.6 BANCO DE DADOS RELACIONAL

Este projeto utiliza uma arquitetura de banco de dados híbrida. O MariaDB (compatível com MySQL) será instalado em um servidor remoto para armazenamento centralizado de dados críticos, como cadastros, ordens de serviço e relatórios. Essa solução centralizada é adotada devido à robustez, escalabilidade e ampla adoção do MariaDB em aplicações corporativas de médio/grande porte. Paralelamente, usamos SQLite como banco de dados local no dispositivo cliente. O SQLite não requer administração e foi projetado para operar em dispositivos embarcados e aplicações offline. Ele armazena temporariamente informações como registros de ordens e logs de operação, permitindo que o sistema continue funcionando mesmo sem conexão. Quando houver conectividade, dados-chave podem ser sincronizados com o servidor remoto, garantindo integridade e segurança da informação. Essa combinação alia a portabilidade do SQLite à centralização e segurança do MariaDB, possibilitando que o sistema funcione de forma eficiente em cenários com conectividade limitada.

2.7 TAURI

O Tauri é um framework open-source para criação de aplicações multiplataforma (desktop e mobile) que combina um front-end web (React, Vue, Angular, TypeScript, etc.) com lógica de back-end em Rust. Sua principal característica é a geração de executáveis muito leves, utilizando o mecanismo de WebView nativo do sistema operacional em vez de embutir um navegador completo. Por exemplo, um aplicativo Tauri pode ter apenas cerca de 600 KB, destacando-se em relação a concorrentes como Electron. Além disso, o Tauri permite reaproveitar tecnologias web no cliente enquanto usufrui da performance e segurança da linguagem Rust no servidor local. Essas qualidades tornam o Tauri adequado para o sistema proposto, pois possibilita desenvolver um aplicativo leve, rápido e seguro para múltiplas plataformas, com código base unificado.

3 METODOLOGIA

Adotou-se uma abordagem sistemática de engenharia de software, inspirada em metodologias ágeis, com ciclo iterativo de entrega contínua. O desenvolvimento do sistema envolveu revisão de literatura, levantamento de requisitos, modelagem, implementação e testes integrados. Em cada iteração, definiram-se tarefas específicas, foi realizada codificação (front-end em TypeScript e back-end em Rust via Tauri), executados testes automatizados e coletado feedback para ajustes. Abaixo, resumimos as principais atividades conduzidas

3.1 LEVANTAMENTO E ANALISE DE REQUISITOS

Elaborou-se o modelo conceitual da aplicação, incluindo diagramas de casos de uso e arquitetura. Um diagrama de casos de uso ilustra os atores (por exemplo, Proprietário e Cliente) e suas interações principais (criação de ordem de serviço, geração de contrato, consulta de status). A arquitetura em camadas define separação entre interface (Angular/TypeScript), lógica de negócio (back-end em Rust via Tauri) e persistência (MariaDB/SQLite). Segundo Bass et al., essa separação modular em camadas facilita flexibilidade e manutenção, permitindo que cada camada evolua independentemente sem comprometer o sistema como um todo.

3.1.1 REQUISITOS FUNCIONAIS

RF001: Cadastrar clientes com nome, telefone e e-mail.

RF002: Criar ordens de serviço com campos de descrição, data, status e responsável.

RF003: Atualizar ordens de serviço existentes.

RF004: Consultar status da ordem por parte do cliente.

RF005: Gerar contrato em PDF com base nos dados da OS.

RF006: Gerenciar usuários com diferentes permissões (administrador, técnico).

RF007: Sincronizar dados locais com banco remoto sempre que houver conexão.

3.1.2 REQUISITOS FUNCIONAIS

RNF001: O sistema deve operar offline utilizando SQLite local.

RNF002: Deve haver autenticação básica para usuários (login/senha).

RNF003: A aplicação deve ser multiplataforma (Linux/Windows).

RNF004: A interface deve ser responsiva e intuitiva.

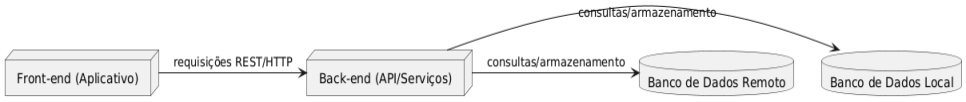
RNF005: O tempo de carregamento inicial não deve ultrapassar 2 segundos.

3.2 TESTES E VALIDACAO

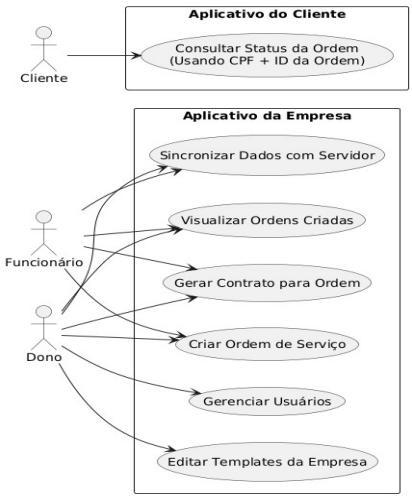
Para cada funcionalidade implementada, serão executados testes unitários e de integração, garantindo o fluxo correto de dados entre front-end, back-end e bancos de dados. Cenários de teste foram criados para simular uso real do software: geração automática de contratos em PDF, criação e atualização de ordens de serviço, sincronização de dados offline/online, etc. Essas validações asseguraram que o sistema atendesse aos requisitos definidos e funcionasse de forma consistente sob condições variadas.

3.3 PROJETO E MODELAGEM DO SISTEMA

Elaborou-se o modelo conceitual da aplicação, incluindo diagramas de casos de uso e arquitetura. Um diagrama de casos de uso ilustra os atores (por exemplo, Proprietário e Cliente) e suas interações principais (criação de ordem de serviço, geração de contrato, consulta de status). A arquitetura em camadas define separação entre interface (Angular/TypeScript), lógica de negócio (back-end em Rust via Tauri) e persistência (MariaDB/SQLite). Segundo Bass et al., essa separação modular em camadas facilita flexibilidade e manutenção, permitindo que cada camada evolua independentemente sem comprometer o sistema como um todo.



- Diagrama de arquitetura



- Diagrama de caso de uso

4 CONCLUSÃO

Este trabalho apresentou o formato e o plano de desenvolvimento de um sistema multiplataforma para gerenciamento de ordens de serviço em empresas de assistência técnica. A solução proposta busca modernizar e simplificar o processo interno de manutenção, oferecendo funcionalidades avançadas como geração automática de contratos, perfis de usuário segmentados e operação offline. Ao combinar tecnologias como Tauri, TypeScript, Rust, MariaDB e SQLite, o sistema prevê aplicações leves e seguras, adequadas à realidade de empresas que nem sempre dispõem de conexão de alta disponibilidade.

Espera-se que o sistema contribua para padronizar processos internos, reduzir erros operacionais e aumentar a produtividade da equipe técnica e administrativa. A arquitetura em camadas facilita manutenção e futuras expansões, como migração para outros bancos ou integração com APIs externas. Como trabalhos futuros, sugere-se implementar testes de usabilidade, avaliar desempenho real em campo e explorar a geração de indicadores gerenciais no dashboard. Assim, esta plataforma tem potencial para aprimorar significativamente a gestão de ordens de serviço, alinhando eficiência operacional às boas práticas de engenharia de software.

REFERÊNCIAS

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. *Software architecture in practice*. 3. ed. Boston: Addison-Wesley, 2012.

HARRINGTON, H. James. *Business process improvement: the breakthrough strategy for total quality, productivity, and competitiveness*. New York: McGraw-Hill, 1991.

IBM. *O que é uma ordem de serviço?* 2023. Disponível em: https://www.ibm.com/br-pt/topics/work-order.

OWENS, Mike. *The definitive guide to SQLite*. 2. ed. Berkeley: Apress, 2010.

PRESSMAN, Roger S.; MAXIM, Bruce R. *Engenharia de software: uma abordagem profissional*. 8. ed. Porto Alegre: AMGH, 2016.

SOMMERVILLE, Ian. *Engenharia de software*. 9. ed. Porto Alegre: AMGH, 2011.