**Projeto de Sistemas distribuídos: Sistema de Gerenciamento de Estacionamento**

[[1]](#footnote-1)Amandio Arnoldo Hoffmann e Vitor Hugo Tavares

**1 RESUMO**

Este artigo descreve o projeto e a implementação de um sistema de gerenciamento de estacionamento, desenvolvido sob os princípios da arquitetura de microsserviços e orientado a eventos. O objetivo principal foi construir uma aplicação robusta, escalável e resiliente, utilizando um ecossistema de tecnologias modernas de computação em nuvem. A solução é composta por serviços independentes containerizados com Docker, incluindo uma API de entrada, um worker para processamento assíncrono, e múltiplos armazenamentos de dados com propósitos distintos (Redis para estado em tempo real e MongoDB para histórico). A comunicação entre os serviços é desacoplada através de um message broker (RabbitMQ)

**2 INTRODUÇÃO**

A crescente complexidade das aplicações web modernas tem exposto as limitações de arquiteturas monolíticas, que concentram toda a lógica de negócio em uma única base de código. Tais sistemas frequentemente enfrentam desafios de escalabilidade, manutenibilidade e resiliência. Em contrapartida, as arquiteturas distribuídas, em especial os microsserviços, surgem como uma alternativa poderosa, promovendo a divisão do sistema em componentes menores, autônomos e com responsabilidades únicas.

O objetivo deste projeto foi projetar, desenvolver e implantar um sistema distribuído funcional, aplicando esses conceitos a um caso de uso prático: o gerenciamento de um estacionamento. A escolha deste tema permitiu explorar desafios comuns como o gerenciamento de estado em tempo real, o processamento de tarefas em segundo plano e a necessidade de persistência de dados históricos.

Este trabalho justifica-se pela necessidade de demonstrar de forma prática a aplicação de um conjunto de tecnologias e padrões que são fundamentais no cenário de desenvolvimento cloud-native. Ao final, o projeto não apenas entrega uma solução funcional, mas também serve como um guia de referência para a construção de sistemas resilientes e escaláveis.

**3 DESENVOLVIMENTO**

O desenvolvimento do projeto foi guiado por decisões arquiteturais que visavam garantir o desacoplamento e a especialização de cada componente do sistema.

**3.1 Arquitetura do Sistema**

Foram adotados múltiplos modelos arquiteturais que, combinados, formam a espinha dorsal do sistema.

* **Arquitetura de Microsserviços:** O sistema foi decomposto em serviços independentes:
  1. **API Gateway (parking-api):** Ponto de entrada para todas as requisições do cliente, responsável por servir a interface web e delegar tarefas complexas.
  2. **Worker (parking-worker):** Serviço de background que executa a lógica de negócio de forma assíncrona, como o cálculo de taxas de estacionamento.
  3. **Bancos de Dados e Broker:** Redis, MongoDB e RabbitMQ foram tratados como serviços independentes, cada um com sua finalidade.
* **Arquitetura Orientada a Eventos (EDA):** A comunicação entre a API e o Worker é mediada por eventos, utilizando o RabbitMQ como message broker. Para a operação de check-out, a API publica um evento car.exited em uma fila, e o Worker, como consumidor, reage a este evento. Isso garante que a API permaneça ágil e não bloqueada por processos demorados.
* **Persistência Poliglota:** Optou-se por utilizar múltiplos bancos de dados, escolhendo a melhor ferramenta para cada tipo de dado:
  + - **Redis:** Um banco de dados em memória, utilizado para armazenar o estado volátil e em tempo real dos veículos atualmente no pátio. Sua alta performance é ideal para consultas rápidas de status.
    - **MongoDB:** Um banco de dados de documentos, utilizado para a persistência de longo prazo dos registros históricos de cada sessão de estacionamento finalizada. Sua flexibilidade de esquema é ideal para armazenar dados transacionais completos.

**3.2 Tecnologias Utilizadas**

* **Linguagem e Framework:** Python 3 com o framework FastAPI, escolhido por sua alta performance, sintaxe moderna e geração automática de documentação.
* **Containerização:** Docker, para empacotar cada serviço e suas dependências em contêineres portáteis e isolados.
* **Orquestração:** O desenvolvimento foi feito com Docker Compose para simplicidade local, com o deploy final projetado e executado em Kubernetes, a plataforma padrão da indústria para orquestração de contêineres em produção.
* **Mensageria:** RabbitMQ, um message broker robusto que implementa o protocolo AMQP e gerencia as filas de comunicação assíncrona.
* **Bancos de Dados:** Redis e MongoDB.

**4 REFERÊNCIAS:**

* + Documentação Oficial do Kubernetes: [kubernetes.io](https://kubernetes.io/docs/home/)
  + Documentação Oficial do Docker: [docs.docker.com](https://docs.docker.com/)
  + Documentação Oficial do Python: [docs.python.org/3/](https://docs.python.org/3/)
  + Documentação Oficial do FastAPI: [fastapi.tiangolo.com](https://fastapi.tiangolo.com/)
  + Documentação Oficial do RabbitMQ: [www.rabbitmq.com/documentation.html](https://www.rabbitmq.com/documentation.html)
  + Documentação Oficial do Redis: [redis.io/docs/](https://redis.io/docs/)
  + Documentação Oficial do MongoDB: [www.mongodb.com/docs/](https://www.mongodb.com/docs/)

1. Aluno do curso de Sistemas de Informação, Universidade para o Desenvolvimento do Alto vale do Itajaí. [↑](#footnote-ref-1)