1. PROPOSTA GERAL

A venda de ingressos para eventos de alta demanda representa um dos cenários mais desafiadores para sistemas web modernos. Quando um artista popular anuncia uma nova turnê, ou quando ingressos limitados são disponibilizados para um evento exclusivo, milhares, ou mesmo milhões, de pessoas acessam simultaneamente a plataforma de venda no exato momento da abertura. Esse fenômeno expõe de forma dramática as limitações de arquiteturas tradicionais e revela a necessidade crítica de soluções distribuídas robustas.

O desafio não é simplesmente lidar com vários acessos, mas sim gerenciar um pico de demanda concentrado em um período, competindo por um recurso escasso e finito (ingressos disponíveis). Este cenário específico gera uma série de problemas técnicos complexos que expõem as fragilidades de arquiteturas tradicionais.

Em arquiteturas monolíticas, todas as requisições convergem para um único servidor ou pequeno cluster de servidores, que rapidamente se tornam insuficientes para processar milhares de solicitações simultâneas. O banco de dados, por sua vez, se transforma em um gargalo, recebendo um volume massivo de operações de leitura e escrita que excedem sua capacidade de processamento. O resultado frequente dessa sobrecarga são erros no serviço, e em casos extremos, indisponibilidade do sistema.

Outro problema é quando múltiplos usuários tentam comprar o mesmo ingresso simultaneamente e o sistema enfrenta um problema clássico de concorrência. Sem controle adequado, o sistema pode processar verificações de disponibilidade em paralelo, levando à venda do mesmo assento para duas ou mais pessoas diferentes. O overselling ocorre quando as verificações de estoque e as operações de venda não são atômicas, ou seja, não formam uma única unidade indivisível de execução. Sob carga extrema e com falhas intermitentes, o sistema pode gerar estados intermediários incorretos, como ingressos marcados como "reservados" que nunca são efetivamente confirmados nem liberados de volta ao estoque.

A partir desse cenário, este projeto tem como objetivo desenvolver uma solução que aborda todos os problemas mencionados acima, utilizando princípios e padrões de Sistemas Distribuídos

A proposta é construir um sistema que garanta alta disponibilidade sob grandes demandas através da técnica de Sharding no MongoDB, fazendo a separação clara entre "aceitar a requisição" (operação rápida) e "processar a transação" (operação assíncrona posterior), que tenha controle de concorrência no banco de dados, evitando condições de corrida, e que adote processamento assíncrono confiável usando filas de mensagens para transformar picos de tráfego em fluxo controlado.

Para atingir os objetivos do projeto, será usado na aplicação o conceito de Arquitetura Orientada a Eventos (Event-Driven Architecture). No processo da compra, a API receberá e validará a requisição, enfileirando-a em um Message Broker. Este, fará o controle de fluxo, enquanto os workers processarão as transações, realizarão operação atômica no MongoDB para verificar e decrementar o estoque.

2. TECNOLOGIAS UTILIZADAS

A aplicação será feita utilizando a linguagem Python 3.10+, devido sua ampla disponibilidade de bibliotecas especializadas, e arquitetura composta por serviços modulares.

2.1. FAST API

Diferentemente de frameworks tradicionais síncronos como Flask ou Django (que utilizam WSGI - Web Server Gateway Interface), FastAPI foi construído nativamente sobre ASGI (Asynchronous Server Gateway Interface), um protocolo moderno que permite processamento assíncrono verdadeiro. A API poderá receber diversas requisições simultâneas de compra de ingressos, e para cada requisição executará operações não-bloqueantes: validar os dados de entrada, consultar o evento no MongoDB, e publicar a mensagem no RabbitMQ. Durante o tempo de espera dessas operações de I/O, o FastAPI processa outras requisições que estão aguardando.

Adicionalmente, FastAPI oferece validação automática de dados através do Pydantic, uma biblioteca Python que implementa validação de tipos em tempo de

execução. Ao invés de escrever código manual para verificar se um campo é obrigatório, se é do tipo correto, ou se está dentro de um intervalo válido, é definido previamente modelos de dados usando type hints do Python.

Por fim, o framework possui documentação automática interativa via OpenAPI/Swagger. Ao iniciar o servidor, automaticamente uma interface é gerada, onde é possível visualizar todos os endpoints disponíveis, seus parâmetros, schemas de entrada e saída, e até mesmo executar requisições de teste diretamente pelo navegador, facilitando o desenvolvimento e testes do sistema.

2.2. MONGODB E A TÉCNICA DE SHARDING

A camada de persistência será implementada com MongoDB, utilizando a técnica de Sharding para distribuir a carga de leitura e escrita entre vários servidores (shards). Essa implementação evitará que o banco de dados se torne um gargalo durante a alta demanda de vendas. Com o uso dessas ferramentas, conseguimos auxiliar no escalonamento horizontal, o que é vantajoso para o projeto, levando em consideração que ele possui alta demanda de dados. Além disso, para um sistema de vendas de ingressos, a baixa latência é crítica, o que se torna uma característica da técnica pela redução da carga de um único servidor.

O MongoDB é um banco de dados não relacional, orientado a documentos e possui uma estrutura de dados flexível e dinâmica. Conseguiremos lidar bem com dados variáveis, reduzir JOINs ao usar documentos e oferecer alta velocidade de escrita, suportando muitos acessos simultâneos.

A infraestrutura dessa camada incluirá os Shards Servers para armazenar os fragmentos dos dados (ex: Ingressos e Transações), Config Servers para armazenar os metadados do cluster juntamente com as definições de como os dados estão distribuídos e o Router (mongos), que servirá como o ponto de entrada único para o Worker, direcionando as consultas para os shards de forma transparente para a aplicação. Todos esses componentes serão conteinerizados para simular servidores reais e distribuídos por instâncias na AWS.

Por fim, para otimizar essa distribuição, será utilizada uma Hashed Shard Key. Essa escolha é estratégica para evitar o surgimento de "Hot Shards" (quando um único

servidor fica sobrecarregado), garantindo que as operações de venda sejam espalhadas de forma uniforme por todo o cluster.

2.3. RABBITMQ

O RabbitMQ será utilizado como o Message Broker (intermediário de mensagens) do sistema. Sua principal função é realizar o desacoplamento entre o recebimento do pedido de compra e o processamento efetivo da transação. Em sistemas distribuídos de alta demanda, como a venda de ingressos, o uso de uma fila é essencial para evitar que picos de tráfego derrubem o servidor de aplicação ou sobrecarreguem o banco de dados.

O sistema operará sob o modelo de mensageria assíncrona. Quando um cliente solicita a compra de um ingresso, o backend em FastAPI valida os dados básicos e publica uma mensagem na fila do RabbitMQ. O usuário recebe uma resposta imediata de "pedido em processamento", liberando o recurso do servidor para o próximo cliente. Logo em seguida, o RabbitMQ armazena essas mensagens de forma ordenada e persistente, garantindo que nenhum pedido seja perdido, mesmo que o serviço de processamento sofra uma breve interrupção. Por fim, as mensagens armazenadas na fila são consumidas por workers dedicados, que processam as ordens de compra e realizam a persistência final no MongoDB Sharded.

O RabbitMQ é usado para garantir entrega confiável e tolerância a falhas no sistema de ingressos. O mecanismo de ACK assegura que uma mensagem só seja considerada concluída após a confirmação de escrita no MongoDB, caso um worker falhe, a mensagem retorna à fila para reprocessamento. Além disso, a persistência de mensagens com filas duráveis evita perda de compras mesmo se o servidor RabbitMQ reiniciar.

Como resultado, o sistema ganha escalabilidade independente, permitindo aumentar apenas o número de workers quando o volume de compras cresce, sem sobrecarregar o FastAPI. O RabbitMQ também atua como um amortecedor de carga, suavizando picos de acesso e evitando que o MongoDB fique sobrecarregado por muitas escritas simultâneas.

2.4. INFRAESTRUTURA - DOCKER E DOCKER COMPOSE

O Docker é uma plataforma de conteinerização que permite encapsular aplicações junto a todas as suas dependências em unidades isoladas chamadas *containers.* Diferente das máquinas virtuais tradicionais, os *containers* compartilham o *kernel* do sistema operacional hospedeiro mas mantêm processos e recursos de rede isolados.

Com o Docker, cada serviço é definido através de um *Dockerfile* que especifica exatamente como construir sua imagem (qual sistema operacional base usar, quais pacotes instalar, quais portas expor). Se um container funciona em uma máquina de desenvolvimento, funcionará identicamente em produção.

No projeto, cada componente será containerizado individualmente. A API FastAPI terá seu próprio container baseado em uma imagem Python oficial, com todas as dependências instaladas via pip. O MongoDB rodará em um container oficial já pré-configurado e otimizado. O RabbitMQ similarmente terá seu container dedicado. Os workers, embora compartilhem código com a API, rodarão em containers separados, permitindo escalabilidade independente.

Docker Compose é uma ferramenta que gerencia aplicações multi-container. Enquanto Docker lida com containers individuais, Docker Compose permite definir e gerenciar sistemas inteiros compostos por múltiplos serviços interconectados através de um único arquivo declarativo em YAML (docker-compose.yml). Este arquivo descreve a topologia completa do sistema distribuído: quais serviços existem, como se comunicam, quais dependências têm, e como devem ser configurados.

3. JUSTIFICATIVA

A integração entre FastAPI, RabbitMQ e MongoDB Sharded no projeto se relaciona com a disciplina de Sistemas Distribuídos nos quesitos de Escalabilidade, disposta através da técnica de Sharding no MongoDB. O projeto demonstra o conceito de distribuição de dados em múltiplos nós. Isso permite que o sistema cresça horizontalmente, dividindo a carga de processamento e armazenamento, evitando que um único servidor se torne um gargalo. Essa arquitetura também fornece transparência de localização, onde a aplicação interage com o cluster como se fosse um banco único.

Outra característica ligada à disciplina se trata do gerenciamento de requisições e alta concorrência. A camada de aplicação desenvolvida em FastAPI utiliza o conceito de processamento assíncrono para operações de leitura e escrita. Isso é essencial em sistemas distribuídos para gerenciar eficientemente milhares de conexões simultâneas sem o bloqueio de recursos, permitindo que a interface permaneça responsiva mesmo sob alta demanda.

Há também o desacoplamento e comunicação indireta através do RabbitMQ, o projeto implementa a comunicação assíncrona baseada em mensagens. Isso remove o acoplamento temporal entre o serviço de recepção (FastAPI) e o de persistência (Worker/MongoDB). O uso de um Message Broker ilustra como sistemas distribuídos gerenciam picos de carga e garantem a resiliência e tolerância a falhas, permitindo que os componentes do sistema operem de forma independente e em velocidades distintas sem perda de dados.

4. EQUIPE E RESPONSABILIDADES TÉCNICAS INDIVIDUALIZADAS

A divisão de tarefas foi estruturada para que cada integrante gerencie camadas específicas que receberam como tema da arquitetura distribuída, colaborando nos pontos de integração.

Milena Mota cuidará da camada de aplicação e interface de mensageria. Seu trabalho incluirá o desenvolvimento da API, responsável pela implementação do backend utilizando FastAPI, criando os endpoints para listagem de eventos e recepção de pedidos de compra. A API será conteinerizada, tendo a criação e configuração do ambiente Docker para a aplicação, garantindo a comunicação via rede interna com o broker de mensagens.

Além disso, é responsável pela integração com RabbitMQ (Produtor), configurando a lógica de publicação de mensagens, garantindo que as solicitações de compra sejam validadas e enviadas corretamente para as filas de processamento.

Mariane Oliveira cuidará da camada de dados, escalabilidade e processamento assíncrono. Isso incluirá a Arquitetura de Dados Sharded, responsável pelo provisionamento e configuração do cluster MongoDB, incluindo a definição dos Shard Servers, Config Servers e do roteador Mongos. Também responsável pela

estratégia de particionamento, definindo e implementando a Shard Key para garantir a distribuição uniforme dos dados e evitar sobrecarga em nós específicos.

Outra responsabilidade é o desenvolvimento do Worker (Consumidor), implementando o serviço que consome as mensagens do RabbitMQ, processa a lógica de negócio final e realiza a persistência atômica no banco de dados. Por fim, a tarefa envolvendo a infraestrutura cloud, configurando as instâncias e rede distribuída na AWS, garantindo que os containers operem em um ambiente de nuvem real.

Há também as responsabilidades compartilhadas, como a modelagem do esquema de dados (JSON/BSON) para eventos e ingressos, definição dos contratos de mensagem (payloads) que trafegam no RabbitMQ e a orquestração final do sistema utilizando Docker Compose para garantir o funcionamento integrado de todos os componentes distribuídos.

5. LINK DO REPOSITÓRIO

O desenvolvimento do projeto será mantido no repositório do GitHub, acessado através do link: https://github.com/marianeoli/ProjetoVendaDeIngressos.git.

REFERÊNCIAS

Docker e Docker Compose um guia para iniciantes. Disponível em: <https://dev.to/ingresse/docker-e-docker-compose-um-guia-para-iniciantes-48k8>.

FASTAPI. FastAPI. Disponível em: <https://fastapi.tiangolo.com/>.

HABBEMA, H. Docker: Compose. Disponível em: <https://medium.com/@habbema/docker-compose-8c16d02585b4>. Acesso em: 20 dez. 2025.

HABBEMA, H. Pydantic. Disponível em: <https://medium.com/@habbema/pydantic-23fe91b5749b>.Acesso em: 20 dez. 2025

TEAM, M. D. Sharding. Disponível em:

<https://www.mongodb.com/pt-br/docs/manual/sharding/>.

RABBITMQ. Messaging that just works — RabbitMQ. Disponível em: <https://www.rabbitmq.com/>.

RAMOS, D. R. Arquitetura Orientada a Eventos e Mensagens - Daniel Rafael Ramos - Medium. Disponível em: <https://medium.com/@danielrafaelramos/arquitetura-orientada-a-eventos-e-mensagens-8ed9d578041c>.