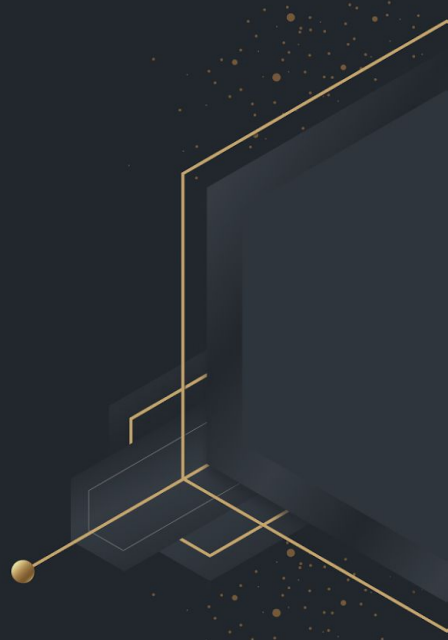
Sobre a Imersão



Sobre a Imersão

A Imersão .NET Expert é um treinamento com duração de 12 horas focado em Arquitetura de Microsserviços com ASP.NET Core, utilizando .NET 6

- **Primeiro dia**
  - **Instrutor**
    - Luis Felipe, Dono e Instrutor @LuisDev, Microsoft MVP
  - **Conteúdos**
    - Arquitetura de Microsserviços
    - Domain-Driven Design
    - Arquitetura Hexagonal
    - Banco de Dados NoSQL
    - Arquitetura orientada a eventos com RabbitMQ

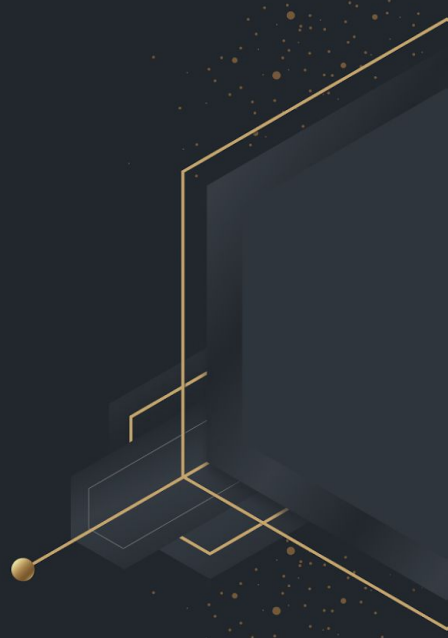
Sobre a Imersão

- **Segundo dia**
  - **Instrutores**
    - Patrick Reinan, Especialista em Arquitetura
    - Henrique Mauri, Arquiteto de Software, Microsoft MVP
  - **Conteúdos**
    - API Gateway com Kong
    - Autenticação com Keycloak
    - Azure DevOps Pipelines com Docker Registry
    - Observabilidade com Elasticsearch, Kibana e Elastic APM

Projeto

- **AwesomeShop:** um projeto para estudos de e-commerce em arquitetura de Microsserviços
- **Composto por:**
  - Customers
  - Products
  - Payments
  - Orders (vai ser implementado)

Microsserviços



Microsserviços

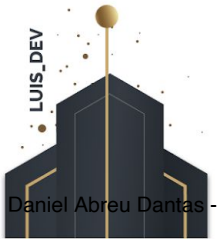
Por muitos anos tentamos desenvolver sistemas utilizando melhores abordagens e ferramentas

- Domain-Driven Design
- Continuous Integration / Continuous Delivery
- Automações de Infra-estrutura (plataformas de virtualizações com provisionamento e redimensionamento)
- Pequenos times de empresas como Google tendo responsabilidade do fluxo completo de desenvolvimento de software
- Adoção de práticas e tecnologias que auxiliam na criação e teste de sistemas mais resilientes em escala, como pela Netflix e Chaos Monkey (por exemplo)

Microsserviços nasceram dessas práticas!

Microsserviços

- O que são Microsserviços?
  - **Serviços pequenos, coesos e desacoplados que trabalham juntos**
  - Muito se fala sobre serem **independentes**, mas no final construímos sistemas e não apenas conjuntos de serviços. Serviços não serão completamente desacoplados ou independentes.
  - **Definição de Sistema:** um conjunto de partes ou dispositivos conectados que operam juntos
  - Serviços sempre terão que interagir com outros para formar um sistema, mas o desacoplamento na integração tem papel essencial para que o sistema não fique frágil



Microsserviços

- Quão pequenos?
  - Sabemos descrever se um sistema é grande ou pequeno, certo?
  - Segundo Jon Eaves (contribuidor do livro Building Microsservices, escrito pelo autor Sam Newman), um microsserviço deve ser pequeno o bastante para ser reescrito em 2 semanas, e autônomo para que possa ser alterado e publicado sem afetar outros serviços

Não deixa de ser uma Arquitetura Distribuída, tendo os desafios existentes desse tipo de arquitetura

Microsserviços

- Desafios
  - Integração e Consistência dos dados
    - REST API? gRPC? Mensageria?
  - Falhas na rede e na comunicação, que podem comprometer o item anterior
  - Sobrecarga no gerenciamento de infraestrutura
    - Múltiplos sistemas sendo publicados de maneira isolada
    - Ferramentas de observabilidade
    - Ferramentas de automação em DevOps
    - Possibilidade de múltiplas tecnologias para aplicações e bancos de dados

Microsserviços vs Monolitos

Daniel Abreu Dantas - contatodanieldantasdev@gmail.com - IP: 172.226.128.45

Microsserviços vs Monolitos

- Big Ball of Mud
  - Software que cresce de maneira descontrolada, resultando em maior custo de manutenção e incidência de bugs
- Não é exclusividade de Monolitos, mas pode ser impulsionado por conta de
  - Múltiplos times atuando em cima da mesma base de código
  - Domínios de negócio complexos
  - Alto acoplamento entre componentes
  - Requisitos não-funcionais conflitantes
- Este problema pode ser lidado através de implementação de microsserviços
  - Reforço: sem seguir as melhores práticas, microsserviços também podem se tornar Big Ball of Mud

Microsserviços vs Monolitos

Característica	Monolitos	Microsserviços
Repositórios	Único	Múltiplos
Tecnologias	Única	Possivelmente Múltiplas
Publicação	Única	Múltiplas
Ponto Único de Falha	Sim	Não
Manutenção	Mais propenso e se tornar BboM	Fácil*

Microsserviços vs Monolitos

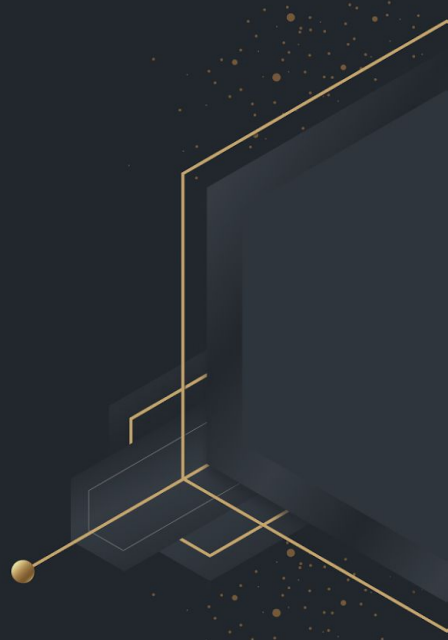
Característica	Monolitos	Microsserviços
Comunicação entre Componentes	In-Process	Rede
Integração	Síncrona	Majoritariamente Assíncrona
Consistência	Imediata	Eventual
Testabilidade (Integração, Aceitação)	Fácil	Difícil
Escalabilidade	Vertical	Horizontal

Principais benefícios

Principais benefícios

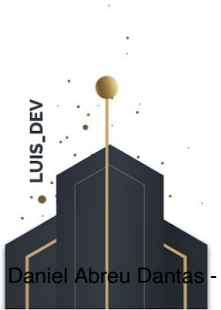
- Resiliência e maior tolerância a falhas
    - Reduzindo os Single Point of Failure
- Heterogeneidade de tecnologias
- Publicações mais rápidas
- Melhor manutenção do código
    - Base de código menor e mais simples, mais fácil de entender, implementar e evoluir
- Otimizado para escalabilidade horizontal, por ser mais rápido de se inicializar

Domain-Driven Design



Domain-Driven Design

- Abordagem de desenvolvimento orientado ao domínio do negócio, alinhando a sua linguagem e termos ao software escrito
- Termo apresentado por Eric Evans no livro "Domain-Driven Design: Tackling Complexity in the Heart of Software"
- NÃO é uma arquitetura, nem força a utilização de uma em específico



Domain-Driven Design

- Assim como é visto com Design Patterns e padrões arquiteturais no geral, é importante não nos tornarmos os famosos "puritanos", que no final se tornam pessoas difíceis de se lidar. Não somente isso: pode impactar na qualidade da solução desenvolvida
- Exemplo: no DDD utilizamos repositórios para tratar persistência de informações. Porém, na prática, podem ocorrer problemas de performance ao utilizar repositório para atualizar informações em lote
    - Atualizar o status de todos alunos de uma turma, ao concluir o semestre, um a um com o repositório

Domain-Driven Design

- O valor principal do Domain-Driven Design está em modelar software que **resolva um problema do negócio**, e não simplesmente **estruturar nosso projeto em X pacotes/pastas**
- É essencial ter contato com especialistas do negócio, visto que na prática a teoria é outra: achismos resultam em re-trabalho, seja através de refatorações ou mesmo de descartes de funcionalidades inteiras que não são úteis para os usuários
- Não buscar o perfeccionismo em uso de padrões. O DDD ataca a complexidade, o inverso do que ocorre em muitos sistemas

Linguagem Ubíqua

Linguagem Ubíqua

- Também conhecida como Linguagem Onipresente, é um termo usado por Erick Evans para descrever uma linguagem compartilhada pelo time
- Quando se fala em "time", estão incluídos desenvolvedores, designers, analistas e especialistas de negócio, gerentes, entre outros membros
- Porém, um cenário muito comum em projetos de software é ter programadores e outros membros distantes do cliente e negócio
    - Isso pode resultar de desinteresse da equipe, ou mesmo da própria estrutura e processo, entre outros fatores

Linguagem Ubíqua

- A Linguagem Ubíqua deve ser a única linguagem utilizada para expressar o modelo, sem ambiguidades ou tradução
- O mesmo "elemento" ou "objeto" podem ter termos diferentes dependendo da equipe (e seu modelo correspondente)!

Linguagem Ubíqua

- Por exemplo, em um cenário de e-commerce, uma pessoa logada pode ser chamada tanto de "usuário", "cliente" ou "solicitante" (ticket de suporte)
    - Nesse caso entramos em temas como contextos delimitados, onde o "cliente" existiria em um contexto de "Vendas" e "Solicitante" em um contexto de "Suporte"
    - Em cada um teriam representações diferentes deles, com as informações específicas de cada um

Linguagem Ubíqua

- Assim como os requisitos do negócio, a Linguagem Ubíqua está em constante evolução e atualização, caminhando junto com o entendimento do time do negócio
    - Infelizmente, em diversos projetos, o entendimento passa a ocorrer quando os prazos de entrega se aproximam, e aí muitos erros em modelagem são descobertos

Entidades

Entidades

- Objeto definido por sua identidade, com valor único
- Não é definida pelos seus atributos, que podem ser atualizados, sem a perda da identidade
- Exemplo: uma pessoa pode ser representada pelo seu CPF, e pode mudar seu endereço ou telefone sem mudar sua identidade
- Identificadas geralmente através de substantivos contidos nas descrições de casos de uso

Entidades

- A nível técnico, é recomendado manter suas propriedades com setters privados, para se ter maior controle de seu estado e regras de negócio, e também evitar inconsistências por conta de alterações por outras partes de código
- Nesse cenário, construtores sem parâmetros junto a propriedades com setters públicos não são recomendados, visto que existe uma grande liberdade se alterar o estado do objeto
- É importante que seja simples inicializar entidades "do jeito certo", seja através de um método Factory interno ou um construtor com assinatura legível

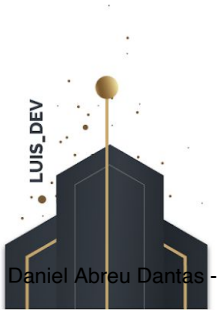
Entidades

- Muitas vezes alguns métodos como Factories e construtores recebem muitos parâmetros e ficam pouco legíveis
- **Data Clumps:** Code Smell relacionado ao uso de grupos de variáveis (como parâmetros para conexão de banco de dados) em diversas partes do código. Esses aglomerados podem (e muitas vezes devem) se tornar classes
    - Como proceder:
        - Extrair classe
        - Substituir onde as variáveis eram utilizadas como parâmetro
    - Reduz tamanho do código e melhora organização e compreensão do código

Value Objects

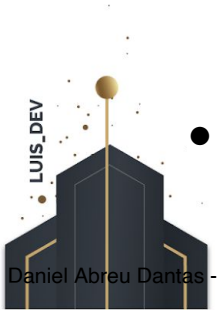
Value Objects

- Objeto definido pelos valores de suas propriedades, e não por apenas uma única
- Ao se comparar a outro, se um valor for diferente, serão considerados objetos diferentes
- Por exemplo, um ponto no plano cartesiano é representado por coordenadas X e Y.



Value Objects

- O exemplo mais famoso é de endereço
- Geralmente um endereço é representado por informações como:
  - Logradouro
  - Complemento
  - CEP
  - Cidade
  - Estado
  - País
- Se um deles for diferente, já não estamos falando do mesmo endereço!



Agregados

Agregados

- Aglomerado de objetos do domínio (entidades e value objects) que podem ser tratados como uma unidade única
- Um dos seus componentes será a raiz do Agregado (Aggregate Root). A raiz do Agregado é muito importante pois garante a integridade do Agregado, tendo quaisquer referências de fora do agregado sendo feitas através dela
- Por exemplo, a entidade **Pedido** poderia ser a raiz do Agregado **Pedido**, que contém outros objetos de domínio como entidades (Cliente, Itens) e Value Objects (Endereço de Entrega e Pagamento, Informações de Pagamento)

Agregados

- Os itens e endereço de pedido não existem sem a raiz do Agregado **Pedido**, e devem estar consistentes entre si, sendo isso garantido pela raiz do **Agregado** que deve controlar as operações feitas
- Geralmente haverá um Repositório por Agregado, já que não é indicado alterar elementos do Agregado diretamente sem passar pela raiz dele

Repositórios

Repositórios

- Padrão utilizado para realização de operações em conjuntos de informações, agnóstico à tecnologia utilizada
- Geralmente associado a um Agregado
- Em aspectos mais técnicos, a definição de sua interface ficaria mais próxima do Core do domínio, e sua implementação da infraestrutura
- Pode ser utilizado como fronteira entre o modelo de persistência e o de domínio, dependendo do banco de dados utilizado

Contextos Delimitados

Contextos Delimitados

- Mapeia um subdomínio para uma solução, contendo suas pecularidades próprias na Linguagem Ubíqua para Entidades, Value Objects, entre outros componentes do domínio
- É um padrão central em Domain-Driven Design, lidando com grandes modelos e times, resultando em diferentes Contextos Delimitados e definindo relacionamentos entre eles
- Resultam de diferentes grupos de pessoas que usam vocabulários diferentes em diferentes partes de uma organização, formando subdomínios

Bounded Context: The conditions under which a particular model is defined and applicable

Eric Evans, "DDD and Microservices: At Last, Some Boundaries!" - QCon London 2016"

Contextos Delimitados

- Alguns objetos de domínio podem existir em múltiplos contextos delimitados, como é o caso do já citado Usuário, Cliente e Solicitante.
    - Solicitante, em um contexto de Suporte poderia conter informações como Id, Nome, CPF, E-mail, Telefone e Lista de Solicitações realizadas, relacionadas a seus referidos Pedidos e/ou Produtos específicos da solicitação
    - Cliente em um contexto de Vendas poderia conter informações como Id, Nome, Data de Nascimento, CPF, Endereços cadastrados, Compras realizadas, e Carrinho Aberto

Contextos Delimitados

- Outro exemplo é em uma aplicação bancária, onde o termo Conta ("Account") pode ter significado diferente em múltiplos contextos
  - Conta Bancária
    - Saldo, Número de Conta, Agência
  - Conta de aplicação Web (para login)
    - Relacionadas a Autenticação, Credenciais
- Sobre o aspecto técnico, um contexto delimitado pode ser mapeado para múltiplos microsserviços. O de Vendas, por exemplo, poderia ter Order, Product/Catálogo, Carrinho de Compra, Customer.

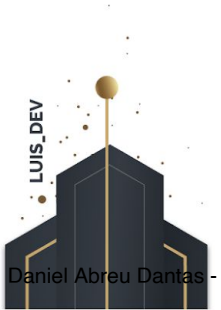
Mapeamentos de Contextos

Mapeamentos de Contextos

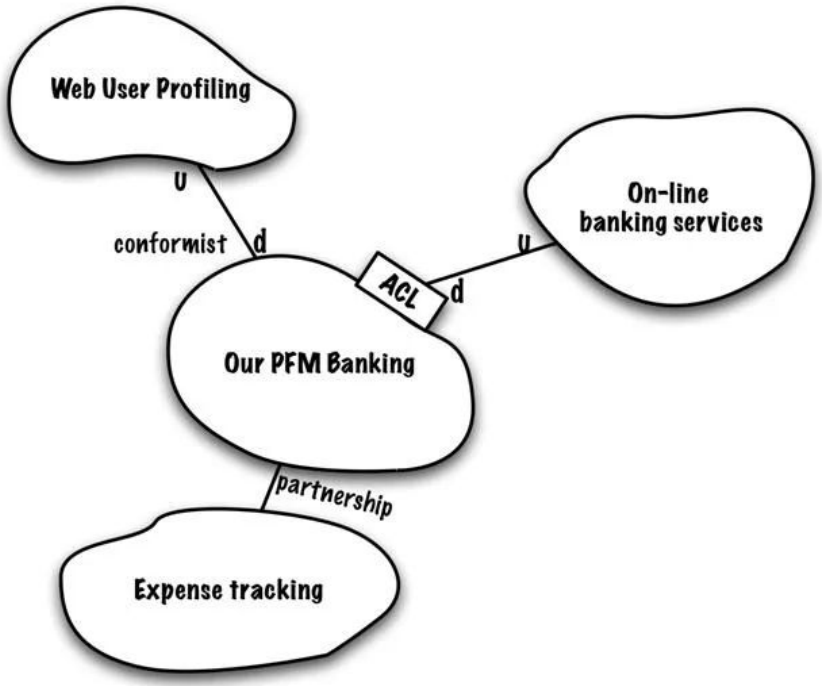
- O mapeamento de contextos auxilia na definição e visualização dos relacionamentos entre contextos delimitados
- Existem alguns tipos de relacionamentos como:
  - **Shared Kernel:** código compartilhado entre contextos/aplicações
  - **Partner:** evolução em conjunto, mas exige maior esforço na parte de gestão
  - **Customer-Supplier:** o Customer (Downstream) depende do Supplier (upstream). Exemplo: Order (Downstream) depende do Warehouse (Upstream)

Mapeamentos de Contextos

- Existem alguns tipos de relacionamentos como:
  - **Conformist:** um Contexto Delimitado depende de outro sem ter grande influência em suas alterações, ficando sujeito a alterações que necessitar fazer devido a alterações no Contexto Delimitado que depende. Exemplo: mudanças de regras em Registro de Boletos pelo banco
  - **Anti-Corruption Layer:** adequado para integração com sistemas legados, é uma camada que adapta quaisquer modificações que ocorram no upstream (em quem depende), para blindar o contexto dessas alterações, como entre formatos diferentes (XML - JSON)

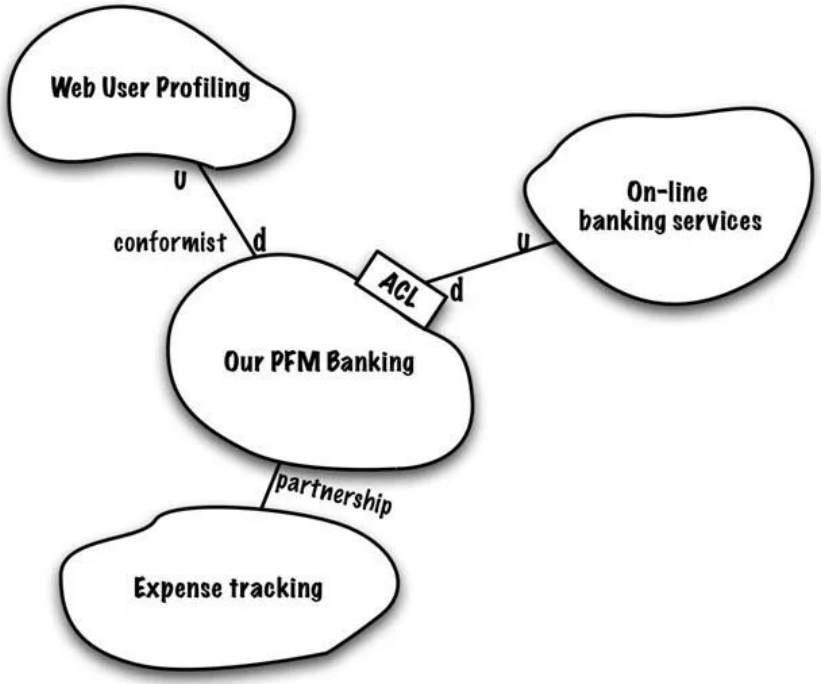


Mapeamentos de Contextos



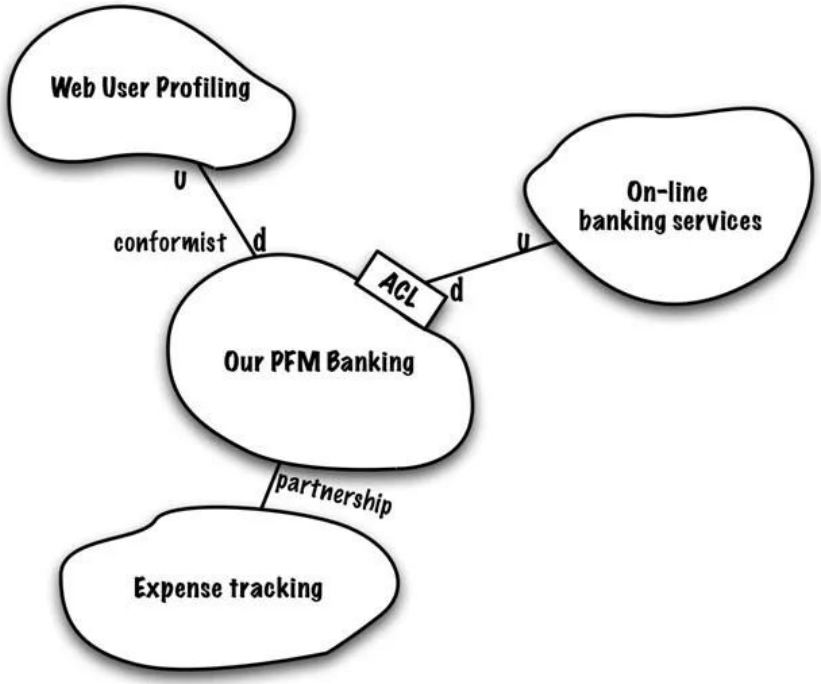
Strategic Domain Driven Design with Context Mapping, 2009 InfoQ, Alberto Brandolini

Mapeamentos de Contextos



- **Banking & On-line Banking Services:** é mantido por um time externo, influencia o Banking (Customer - Supplier), forçando alterações caso mude sua API, por conta disso foi inserida uma Anti-Corruption Layer
- **Banking & Expense:** partnership, por terem objetivos comuns e estarem no mesmo nível

Mapeamentos de Contextos

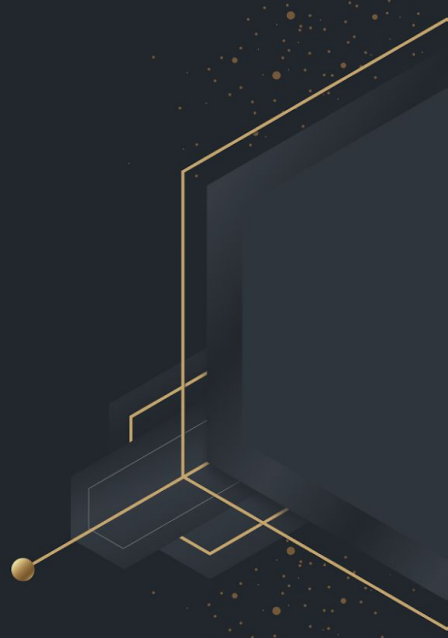


- **Banking & Web User Profile:** como o Profiling é realizado por um módulo externo, existe uma relação de conformist e o Banking precisa se adaptar às alterações do Profiling

Criando o Microsserviço

Como Estruturar

Arquitetura Hexagonal



Daniel Abreu Dantas - contatodanieldantasdev@gmail.com - IP: 172.226.128.45

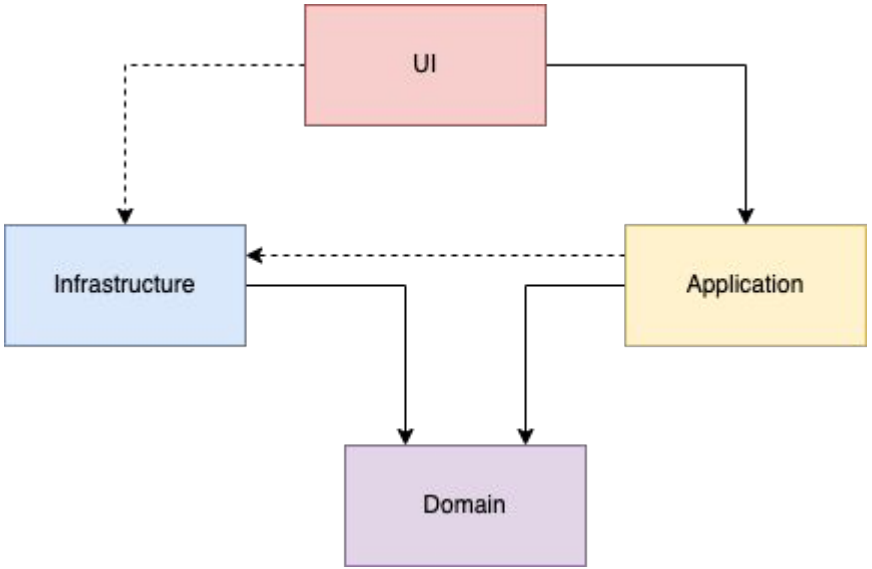
Arquitetura Hexagonal

- Arquiteturas em camadas (Layered Architectures) tem como objetivo separar uma aplicação em diferentes camadas, onde cada uma teria classes e módulos com responsabilidades similares (coesas)
- Arquitetura Hexagonal, ou Ports and Adapters, é uma arquitetura que tem como foca desacoplar a implementação de casos de uso dos detalhes externos
- Implementação de casos de uso interagem com serviços e entidades, e regras de negócio são descritas através de Entidades, Agregados, Value Objects, e Serviços de Domínios, por exemplo

Arquitetura Hexagonal

- Ports se referem à forma de comunicação externa, como um endpoint definido através de um Controller e Action
- Adapters contém implementações de acesso a dados, APIs externas, serviços de infraestrutura, etc
- Em seu livro, Eric Evans propõe uma arquitetura em 4 camadas, existindo uma separação entre a camada Domain e UI, Application e Infrastructure
    - Domain é totalmente independente de camadas e frameworks
    - Application depende na Domain, mas também é independente de frameworks e tecnologias de bancos de dados
    - A UI depende da Application e utiliza a Infrastructure de maneira indireta (DIP)

Arquitetura Hexagonal



Arquitetura Hexagonal

- Princípio de Inversão de Dependência tem um papel fundamental em desacoplar a camada Application da implementação de Repositórios e outros serviços com implementação na Infrastructure que ela possa usar
    - High-level modules should not depend on low-level modules. Both should depend on abstractions.

Arquitetura Hexagonal

- Camada UI (API, no nosso caso)
  - Definição dos endpoints (Controllers e Actions)
  - Filters e Middlewares
- Camada Application
  - Use Cases
  - Input e Output
  - Presenter*
- Camada Infrastructure
  - Repositórios (implementações)
  - Acesso a APIs e serviços Externos (Cloud, serviços de infraestrutura, etc)

Arquitetura Hexagonal

- Camada Domain
  - Entidades
  - Agregados
  - Value Objects
  - Domain Services
  - Eventos de domínio
  - Repositórios (interface)
  - Exceções de domínio

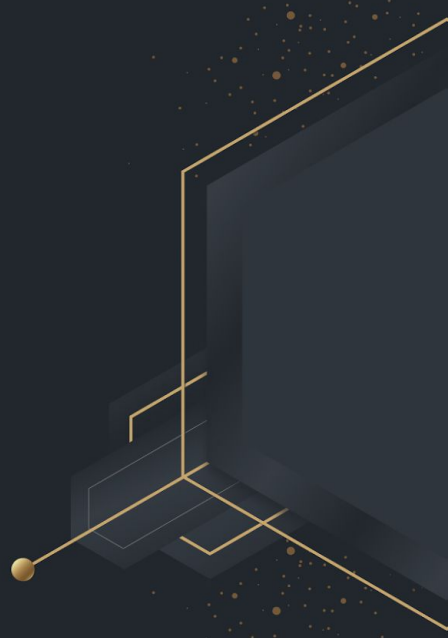
Sobre a Arquitetura Limpa

Sobre a Arquitetura Limpa

- Na Arquitetura Limpa, temos diversos conceitos similares aos aplicados na Arquitetura Hexagonal
- As camadas são fundamentalmente as mesmas, mas a diferença principal está na implementação dos casos de uso na camada de Aplicação
    - Na Arquitetura Limpa existe o conceito de serviços de aplicação
    - Na Arquitetura Hexagonal, existe o conceito de Use Case
- Na Arquitetura Limpa é comum nomear a camada Domain de Core
- Mas lembre-se: evitar ser puritano quanto a isso, o que importa é construir uma arquitetura flexível a mudanças, com boa manutenção e testabilidade

Arquitetura Hexagonal na Prática

Bancos de Dados NoSQL: MongoDB



Daniel Abreu Dantas - contatodanieldantasdev@gmail.com - IP: 172.226.128.45

Banco de Dados NoSQL: MongoDB

- Bancos de Dados NoSQL são bancos de dados não-tabulares, que armazenam dados de maneira diferente a bancos de dados relacionados
- Nasceram no final dos anos 2000s, em um momento em que o custo de armazenamento foi sendo reduzido
- Desenvolvedores começaram a se tornar o principal custo no desenvolvimento de software, e por conta disso os bancos de dados NosQL são otimizados para sua produtividade

Banco de Dados NoSQL: MongoDB

- A quantidade de dados que as aplicações precisavam armazenar e interagir aumentou, tendo grande diversidade de formatos e tamanhos
  - Estruturados
  - Não-Estruturados
- Não somente isso, por conta da natureza ágil de projetos e seus requisitos, existe a necessidade de constantemente atualizar requisitos, impactando os modelos de dados
- Bancos de dados relacionais começaram a apresentar maior dificuldade em serem utilizados por conta do schema ter de ser definido de antemão

Banco de Dados NoSQL: MongoDB

- Os tipos principais de bancos de dados NosQL são:
  - Chave-Valor (Redis)
  - Documentos (MongoDB, RavenDB)
  - Grafos (Neo4J)
  - Orientados a colunas (Cassandra)

Banco de Dados NoSQL: MongoDB

- Finalmente, alguns bancos de dados NoSQL como o MongoDB oferecem a capacidade de distribuir dados, e escalar horizontalmente, utilizando sharing ou replica sets
    - **Sharding:** espalhar dados ao longo de múltiplos nós, melhorando a performance nas escritas ao ocorrerem em apenas um dos nós e seus dados armazenados
    - **Replica Sets:** duplicação de dados ao longo de múltiplos nós, porém pode apresentar problemas ao tentar manter consistência em cenários de muitas escritas, que seriam propagadas entre os membros do replica set

Sobre o MongoDB

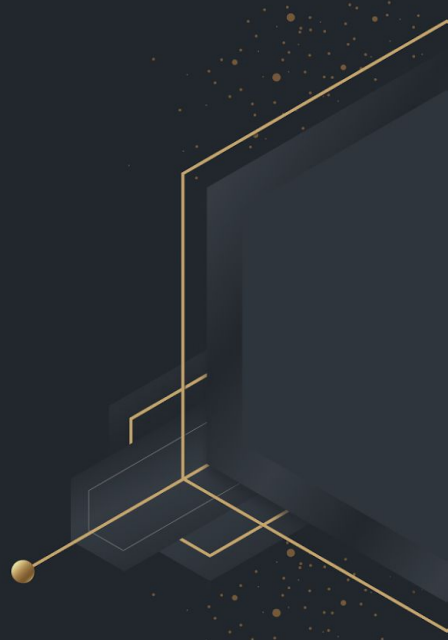
Sobre o MongoDB

- Banco de Dados NoSQL orientado a documentos, com um formato semelhante ao JSON (chamado BSON)
- Os documentos são armazenados em coleções (collections), sendo análogas às tabelas em bancos de dados relacionais
- É um banco de dados de alta performance, e por ter suporte a objetos relacionados facilita a modelagem de dados

Sobre o MongoDB

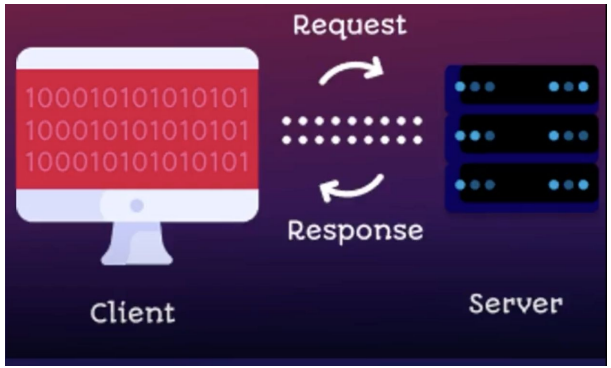
- O pacote oficial para .NET é o MongoDB.Driver, de fácil utilização
- É possível armazenar um Agregado completo como documento, por ele ter suporte a objetos relacionados
- A chave primário de um documento é o campo _id (por convenção, a propriedade Id da classe é usada), e atualizações são feitas no objeto inteiro

Arquitetura Orientada a Eventos



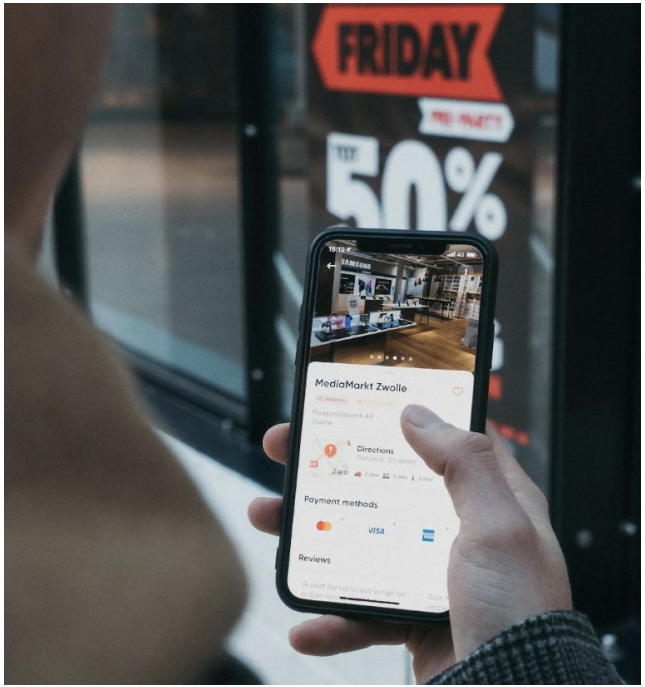
Comunicação Tradicional

- O jeito tradicional de comunicação entre sistemas é o chamado Request-Response
- Nele, um cliente (computador ou aplicação) realiza uma requisição, e o servidor responde com os dados pedidos
- Enquanto o servidor tiver a capacidade de responder as requisições em tempo hábito, esse modelo atende muito bem



Comunicação Tradicional

1. O usuário adiciona itens ao carrinho e decide finalizar a compra
2. A página de checkout carrega
3. O usuário preenche os dados de pagamento
4. Devido a uma instabilidade do serviço de pagamento, a requisição falha e o pedido não é concluído!
5. Mesmo os dados de compra estando corretos, o usuário recebe a mensagem de erro e desiste da compra!



Arquitetura Orientada a Eventos

- Arquitetura de software onde a estrutura é composta por:
  - Captura
  - Comunicação
  - Processamento
  - Persistência
- ... De eventos

Arquitetura Orientada a Eventos

- Um evento, derivado da mesma nomenclatura no Domain-Driven Design, é uma ocorrência significativa do sistema, como uma alteração de uma entidade
- Alguns exemplos são
  - Compra em um E-Commerce
  - Cadastro de um usuário
  - Pagamento de uma assinatura
  - Publicação de um texto

Arquitetura Orientada a Eventos

- Baseado nesses eventos, algumas ações podem ser realizadas, como:
  - Notificação por e-mail ou SMS ao cliente
  - Sincronização de dados com outro sistema
  - Notificação de inscritos em uma newsletter
  - Próxima tarefa do fluxo, como o processamento de pagamento de um pedido ou separação de estoque

Sobre Mensageria

Mensageria

- Devido a comunicação síncrona dificultar a resiliência em cenários de Microsserviços (exigindo padrões como o Retry e o Circuit Breaker), a assíncrona geralmente é preferida
- Utilizando mensageria, é possível publicar eventos como mensagens, contendo os dados relevantes e necessários, a um Message Broker, por exemplo, que é um intermediário na comunicação assíncrona
- Um consumidor (ou inscrito) pode então retirar ou receber a mensagem e processá-la, resultando em uma ação de Acknowledgement (ou Ack), confirmando o processamento da mensagem ao Message Broker

Mensageria

- Um Message Broker, entre outras funções, gerencia filas e tópicos, e garante que as mensagens que estão neles sejam entregues

Mensageria

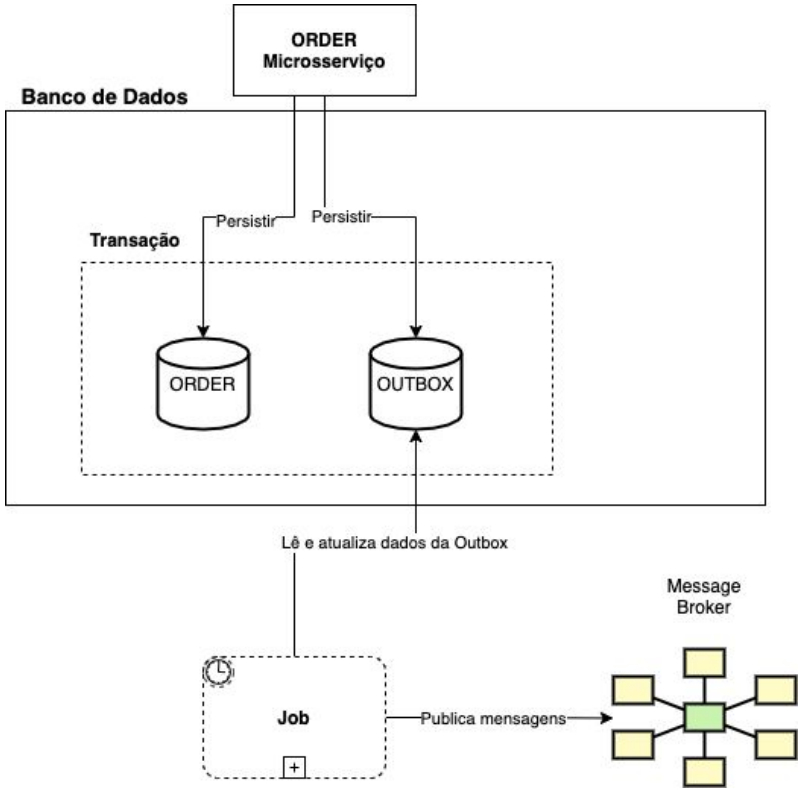
- Benefícios
  - Maior resiliência
  - Menor acoplamento
  - Maior facilidade de integração em questões de esforço de desenvolvimento
- Desafios
  - Tratamento de eventos duplicados
  - Mapeamento e Alinhamento de fluxos de processamentos

Padrão Outbox

Padrão Outbox

- Um padrão frequentemente utilizado junto com a Arquitetura Orientada a Eventos é o Padrão Outbox
- Seu objetivo é garantir a consistência de operações e sua posterior publicação de mensagem ao Message broker, afinal, ele pode sair do ar por alguma falha de infraestrutura ou rede
- Nele, a mensagem a ser publicada é persistida em uma coleção de dados, e um serviço recorrente realiza a sua publicação de maneira independente da operação que originou o evento

Padrão Outbox



Padrão Outbox

- Um passo a passo para implementar esse padrão é:
  - Criar uma tabela ou coleção Outbox, com os dados do evento
  - Persistir eventos de integração na coleção Outbox, e entidades/agregado de maneira transacional
  - Implementar um serviço recorrente, que vai buscar os eventos que não foram publicados e publicá-los no Message Broker, e finalmente marcá-los como publicados na tabela Outbox

Sobre o RabbitMQ

Sobre o RabbitMQ

- O RabbitMQ está entre os Message Brokers mais populares, e tem uma grande quantidade de bibliotecas oficiais com suporte a diversas linguagens/frameworks
- Entre suas principais funcionalidades estão:
  - Tópicos
  - Deadletter Queues
  - Agendamento de Entregas
  - Envio em Batch
  - Transações
  - Deduplicação

Sobre o RabbitMQ

- Fila é o principal conceito em mensageria junto a mensagem, e representa uma estrutura onde as mensagens são armazenadas e consumidas
- Entre suas principais características estão:
  - **Durável:** a fila segue existindo mesmo que o broker reinicie
  - **Auto-Delete:** quando a fila chegue a ter apenas um consumidor e ele se desinscrever, a fila é apagada
  - **Exclusiva:** a fila é utilizada por apenas uma conexão, sendo apagada quando essa conexão for encerrada

Sobre o RabbitMQ

- Outros conceitos importantes de se entender são os de Exchange e Routing Key
    - **Exchange:** agentes responsáveis por rotear as mensagens para filas, utilizando atributos de cabeçalho, routing keys ou bindings
    - **Binding:** conexão utilizada para configurar uma relação entre uma fila e uma exchange
    - **Routing Key:** é um atributo adicionado ao cabeçalho da mensagem, servindo como um "endereço" que o exchange poderá decidir como rotear a mensagem ao definir um Binding

Tipos de Exchange

Tipos de Exchange

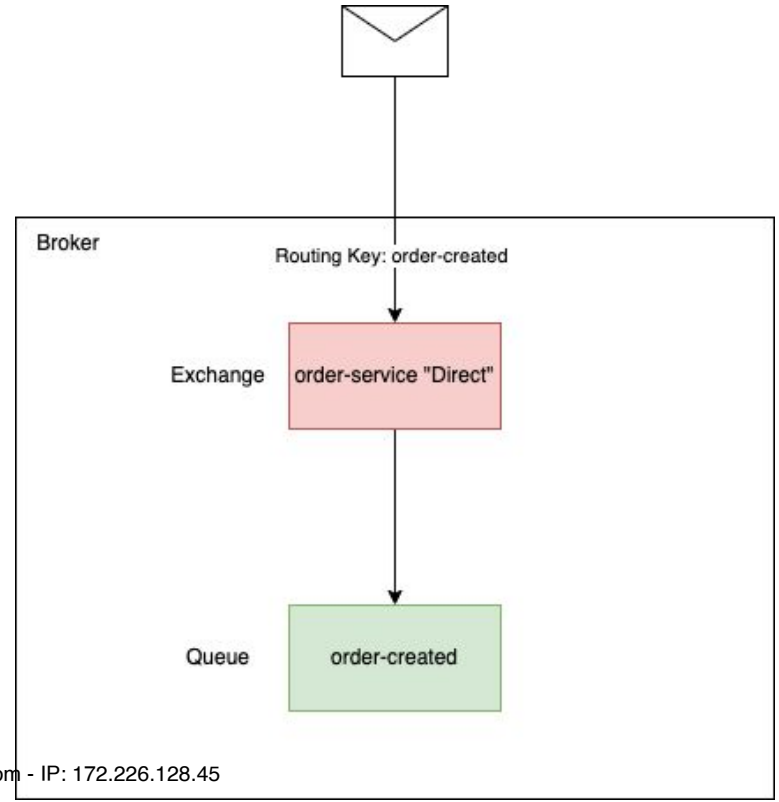
- Direct
- Default
- Topic
- Fanout

Tipos de Exchange

- Direct
  - Envia mensagens para a fila que exatamente encaixa com a routing key

Tipos de Exchange

- Direct

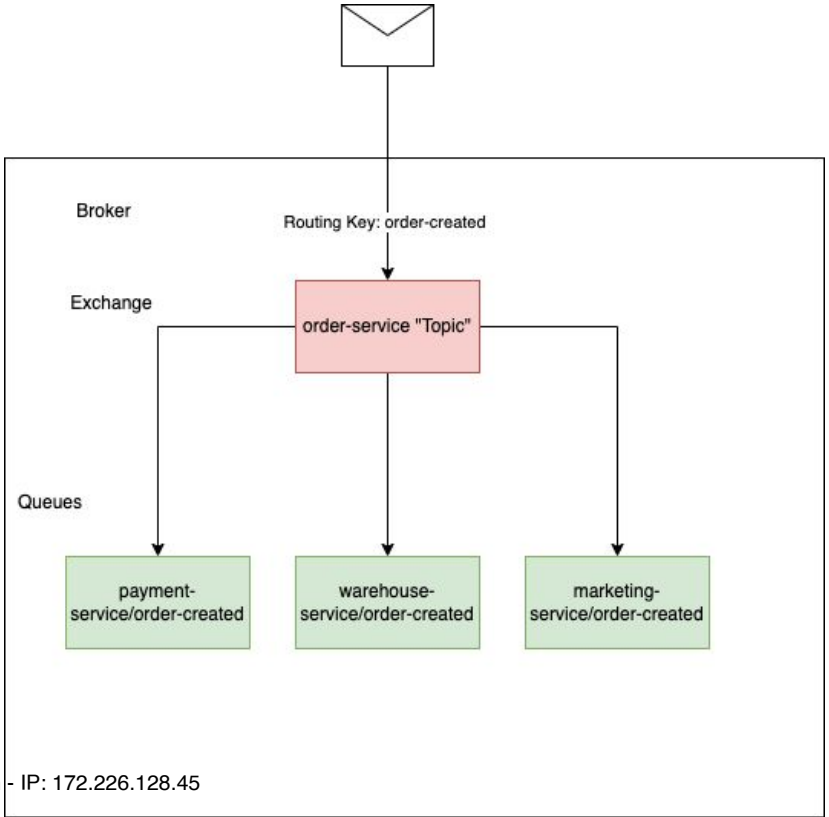


Tipos de Exchange

- Default
  - É uma exchange de tipo "Direct", sem nome "", que todas filas criadas se registram com a routing key igual ao seu nome
- Topic
  - Roteia mensagens para filas baseado na routing key
  - Permite a implementação de padrão publish/subscribe, ou seja, é possível rotear uma mensagem para múltiplas filas

Tipos de Exchange

- Topic

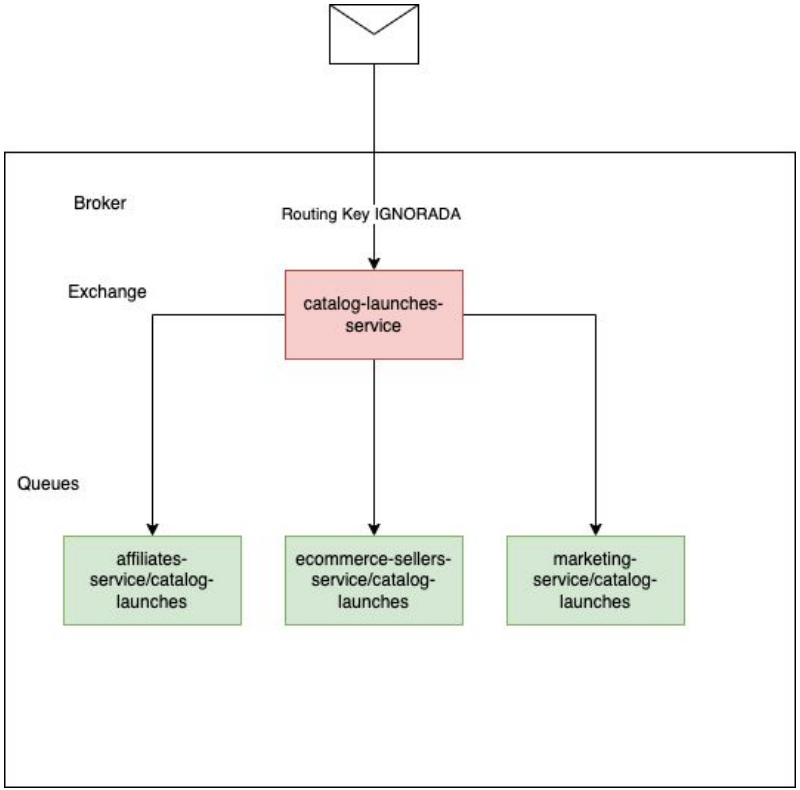


Tipos de Exchange

- Fanout
  - Copia e roteia todas mensagens recebidas para todas filas que estão registradas na Exchange, independente de routing keys ou padrões definidos
  - São úteis quando se necessita enviar a mesma mensagem para 1 ou mais filas de consumidores que irão processar de maneira diferentes

Tipos de Exchange

- Fanout



Publicando eventos

Consumindo Eventos

Encerrando o dia 1

Encerrando o dia 1

- Certificados serão disponibilizados após a segunda aula
- Quem não tiver enviado o usuário do Github, me enviar para eu adicionar na Organization do curso, para ter acesso aos repositórios